Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2012

Ricardo Portugal

Media Gateway utilizando um GPU

Media Gateway using a GPU

"You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat."

— Albert Einstein

Ricardo Portugal

Media Gateway utilizando um GPU

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de Diogo Gomes, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e António Neves, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Dr. Rui Aguiar
Professor associado com agregação da Universidade de Aveiro (por delegação do Reitor da Universidade de Aveiro)

vogais / examiners committee

Prof. Dr. Diogo Gomes
Professor auxiliar convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

Prof. Dr. António Neves
Professor auxiliar convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (co-orientador)

agradecimentos / acknowledgements

Agradeço antes de mais à minha família por todo o apoio incondicional ao longo dos anos, e pelo sentido de responsabilidade, honestidade e de cultura que me transmitiram.

Aos meus amigos, em especial a Ana Zita de Sousa, Inês Albuquerque, João Araújo, João Davim, Marcelo Lebre, Marco Oliveira, Nuno Santos, Pedro Goucha Francisco, Rui Abreu Ferreira, e Rui Carvalho por terem sido tão pacientes comigo ultimamente.

Aos meus orientadores, assim como ao Professor João Paulo Barraca (colaborador), não só por todo o apoio na elaboração desta dissertação, mas por serem professores, pessoas e profissionais acessíveis e competentes. Enquanto aluno, fico extremamente satisfeito por ver estas qualidades na nova geração de professores. Agradeço também a todo o grupo que constitui o Advanced Telecommunications and Networks Group (ATNoG), no Instituto de Telecomunicações - Aveiro.

Finalmente agradeço a todas os músicos que fizeram a "banda sonora" desta dissertação, em particular The Beatles, The National e Yann Tiersen.

Palavras-Chave

Media Gateway, IMS, Redes de Nova Geração, Compressão de voz, DSP, GPU, GPGPU, CUDA, OpenCL, Asterisk

Resumo

Vivemos a convergência entre redes como a Internet e as redes fixas e móveis com as Redes de Nova Geração, através de plataformas como o IMS.
A computação paralela ganhou popularidade junto do utilizador comum, tanto com os processadores multi-core, como com os GPUs (através de APIs como CUDA e OpenCL), a preços relativamente acessíveis.
Esta dissertação propõe estudar a possibilidade de desenvolver uma Media Gateway em software utilizando *software open source* existente, como o Asterisk. Propõe ainda estudar a aceleração da conversão entre formatos de voz (*transcoding*) com CUDA, de modo a utilizar apenas componentes *off-the-shelf*, ao invés de recorrer a equipamentos dispendiosos com DSPs e ASICs.
Utilizando o *software open source* Asterisk, é possível construir uma Media Gateway IMS. Devido à inexistência de uma interface H.248/Megaco no Asterisk, assim como software que implemente as funções SGW, MGCF e BGCF, é mais simples a implementação de toda a PSTN/CS Gateway do que uma Media Gateway. Para mais, o IMS utiliza uma versão extendida do protocolo SIP, mas esse problema pode ser resolvido com uma solução temporária que prevê o descarte dos *headers* SIP provenientes do IMS.
Em CUDA, os algoritmos devem satisfazer alguns requisitos para conseguir ganhos de performance face às implementações em CPU, permitindo frequentemente ganhos até duas ordens de magnitude. CUDA explora um tipo de paralelismo designado de paralelismo de dados (*data parallelism*).
Na tradução entre formatos de voz não é possível ter paralelismo de dados no âmbito de apenas um canal; no entanto pode ser criado paralelismo de dados processando tramas de vários canais em simultâneo. Os codecs abordados recorrem a operações aritméticas com quantização, filtragem, operações com matrizes, e *codebooks*. Uma vez que o acesso às tabelas, necessário para quantização e *codebooks* é aleatório, já que depende, em todas as fases, do valor da trama a ser processada, e devido à utilização de inteiros, os algoritmos não cumprem os requisitos para uma melhoria de performance em relação às implementações em CPU. Outros problemas potenciais, como as operações aritméticas com saturação e o armazenamento necessário para os *codecs*, dificilmente permitirão alguma aceleração, embora estes problemas não tenham sido testados. Por outro lado seria necessária uma modificação profunda no Asterisk para suportar o modelo de *threading* ideal para a utilização de GPUs.
Por estes motivos, apesar de ser possível implementar uma Gateway com algumas limitações utilizando o Asterisk, a utilização de GPUs para *transcoding* não é adequada, sendo preferível a utilização de DSPs para este propósito.

Keywords Media Gateway, IMS, Next-Generation Network, Speech Coding, Speech Compression, DSP, GPU, GPGPU, CUDA, OpenCL, Asterisk

Abstract

Nowadays, we are living the convergence of networks such as the Internet and fixed and cellular networks, with the Next Generation Networks, through platforms such as IMS.

Parallel computing has gained popularity with the end-user, both in multi-core processors, and in GPUs (using APIs such as CUDA and OpenCL), at a fairly low cost.

This dissertation analyzes the possibility of developing a software Media Gateway using existing open source software, such as Asterisk. Furthermore, it proposes the acceleration of speech format translation (transcoding) with CUDA, employing only off-the-shelf components, instead of expensive devices that are equipped with DSPs and ASICs.

It is possible to build an IMS Media Gateway using the Asterisk open source software. Due to the absence of an H.248/Megaco interface in Asterisk, but also the lack of software that can implement the SGW, MGCF, and BGCF functions, it is simpler to implement the whole PSTN/CS Gateway than just the Media Gateway. In addition, IMS uses an extended version of the SIP protocol, but that problem can be solved with a temporary solution, by discarding the SIP headers from IMS.

In CUDA, algorithms must satisfy a series of requirements in order to get performance gains over the CPU implementations, often accelerating them by two orders of magnitude. CUDA exploits a type of parallelism called data parallelism.

It is not possible to have data parallelism in one channel while transcoding; however, it can be created by processing frames belonging to different channels simultaneously. The codecs addressed in this text use arithmetic operations with saturation, quantization, filtering, matrix operations, transforms, and codebooks. The access to tables, required in quantization and codebooks, is random, as it depends on the frame value, and the usage of integers limits the kernel performance; hence the algorithms do not match the requirements needed for a performance achievement over the CPU implementations. Other potential problems, such as arithmetic operations involving saturation, and storage required by the codecs, will hardly allow a speech codec to be accelerated using GPUs, but these were not tested due to time constraints. Furthermore, Asterisk would require extensive modifications in order to support the ideal threading model for the GPU usage.

Because of the aforementioned reasons, although it is possible to implement, within certain limitations, a Gateway with Asterisk, GPUs are not suitable for transcoding, and DSPs assist much better this process.

Contents

List of Figures

List of Tables

List of Acronyms

3G Third Generation

3GPP 3rd Generation Partnership Project

3GPP2 3rd Generation Partnership Project 2

AAA Authentication, Authorization, and Accounting

AAC Advanced Audio Coding

ABR Average Bit Rate

ACD Automatic Call Director

ACELP Algebraic-Code-Excited Linear Prediction

ADC Analog-to-Digital Converter

ADPCM Adaptive Differential Pulse-Code Modulation

ALU Arithmetic Logic Unit

AMR Adaptive Multi-Rate

AMR-NB Adaptive Multi-Rate Narrowband

AMR-WB Adaptive Multi-Rate Wideband

AMR-WB+ Extended Adaptive Multi-Rate - Wideband

ANSI American National Standards Institute

API Application Programming Interface

APP AMD Accelerated Parallel Processing

APU Accelerated Processing Unit

AS Application Server

ASIC Application-Specific Integrated Circuit

ATM Asynchronous Transfer Mode

AuC Authentication Centre

AUMDF Alternatively Updated MDF

B2BUA Back-to-Back User Agent

BGCF Breakout Gateway Control Function

BGP Border Gateway Protocol

BICC Bearer Independent Call Control

BLIT Bit-Block Image Transfer

BOINC Berkeley Open Infrastructure for Network Computing

BRI Basic Rate Interface

CAMEL Customized Applications for Mobile networks Enhanced Logic

CDMA Code Division Multiple Access

CELP Code Excited Linear Prediction

Cg C for Graphics

CGMA Compute to Global Memory Access

CLI Command-Line Interface

CNG Continuous Noise Generator

CPU Central Processing Unit

CS Circuit Switching

CS-ACELP Conjugate-Structure Algebraic-Code-Excited Linear Prediction

CSCF Call/Session Control Function

CTM Close To Metal

CU Compute Unit

CUDA Compute Unified Device Architecture

DAC Digital-to-Analog Converter

DAHDI Digium Asterisk Hardware Device Interface

DBMS Database Management System

DC Direct Current

DCT Discrete Cosine Transform

DFT Discrete Fourier Transform

DMA Direct Memory Access

DNS Domain Name System

DoD Department of Defense

DPCM Differential Pulse-Code Modulation

DRAM Dynamic Random Access Memory

DS-0 Digital Signal 0

DSC Digital Signal Controller

DSL Digital Subscriber Line

DSP Digital Signal Processor

DTMF Dual-Tone Multi-Frequency Signaling

DTX Discontinuous Transmission

DUP Data User Part

ECC Error Correcting Code

EIB Element Interconnect Bus

ENUM E.164 NUmber to URI Mapping

ETSI European Telecommunications Standards Institute

FDM Frequency-Division Multiplexing

FFT Fast Fourier Transform

FHoSS Fokus Home Subscriber Server

FIR Finite Impulse Response

FLOPS Floating Point Operations Per Second

FMA Fused Multiply-Add

FMAC Fused Multiply-Accumulate

FPGA Field-programmable Gate Array

FPU Floating-Point Unit

FXO Foreign eXchange Office

FXS Foreign eXchange Station

GCC GNU Compiler Collection

GLSL OpenGL Shading Language

GNU GNU's Not Unix!

GPGPU General-Purpose computation on Graphics Processing Units

GPL General Public License

GPRS General Packet Radio Service

GPU Graphics Processing Unit

GSM Global System for Mobile Communications

GSM-EFR GSM Enhanced Full Rate

GSM-FR GSM Full Rate

GSM-HR GSM Half Rate

GUI Graphical User Interface

HDL Hardware Description Language

HLR Home Location Register

HLSL High Level Shading Language

HPC High-Performance Computing

HSS Home Subscriber Server

HTTP Hypertext Transfer Protocol

I-CSCF Interrogating-Call/Session Control Function

I/O Input/Output

IAX Inter-Asterisk eXchange

IBCF Interconnection Border Control Function

ICE Interactive Connectivity Establishment

IDCT Inverse Discrete Cosine Transform

IDFT Inverse Discrete Fourier Transform

IDL Interactive Data Language

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IFFT Inverse Fast Fourier Transform

IIR Infinite Impulse Response

IMPI IP Multimedia Private Identity

IMPU IP Multimedia Public Identity

IM-MGW Media Gateway

IM-SSF IP Multimedia Service Switching Function

IMS IP Multimedia Subsystem

IMS-MGW Media Gateway

IN Intelligent Networks

IP Internet Protocol

IPC Inter-Process Communication

IPsec Internet Protocol Security

IPTV Internet Protocol Television

IS-IS Intermediate System To Intermediate System

ISA Instruction Set Architecture

ISDN Integrated Services Digital Network

ISP Immittance Spectrum Pairs

ISUP ISDN User Part

ITU International Telecommunication Union

ITU-T International Telecommunication Union - Telecommunication Standardization Sector

IVR Interactive Voice Response

LAN Local Area Network

LAR Log Area Ratio

LEC Local Exchange Carrier

LGPL Lesser General Public License

LMS Least Mean Squares

LP Linear Prediction

LPC Linear Predictive Coding

LPCM Linear Pulse-Code Modulation

LSF Line Spectral Frequencies

LSP Line Spectrum Pair

LTP Long Term Prediction

MAC Multiply-Accumulate

MAP Mobile Application Part

MDF Multidelay Block Frequency Domain Adaptive Filter

MELP Mixed-Excitation Linear Prediction

MG Media Gateway

MGC Media Gateway Controller

MGCF Media Gateway Control Function

MGCP Media Gateway Control Protocol

MGW Media Gateway

MIMD Multiple Instruction, Multiple Data

MIPS Million Instructions Per Second

MISD Multiple Instruction, Single Data

MOS Mean Opinion Score

MP Multiprocessor

MP3 MPEG-1 or MPEG-2 Audio Layer III

MPEG Moving Picture Experts Group

MPI Message Passing Interface

MPMD Multiple Program Multiple Data

MRF Media Resource Function

MRFC Media Resource Function Controller

MRFP Media Resource Function Processor

MTP Message Transfer Part

NaN Not a Number

NAT Network Address Translation

NGN Next-Generation Network

NUMA Non-Uniform Memory Access

OC Optical Carrier

OpenCL Open Computing Language

OpenGL Open Graphics Library

OpenMP Open Multi-Processing

OSA Open Service Architecture

OSPF Open Shortest Path First

P-CSCF Proxy-Call/Session Control Function

PBX Private Branch Exchange

PCI Peripheral Component Interconnect

PCIe Peripheral Component Interconnect Express

PCM Pulse-Code Modulation

PDA Personal Digital Assistant

PE Processing Element

PIN Personal Identification Number

PLMN Public Land Mobile Network

POSIX Portable Operating System Interface for Unix

POTS Plain Old Telephone Service

PPE Power Processing Element

PRI Primary Rate Interface

PSTN Public Switched Telephone Network

PTX Parallel Thread Execution

QMF Quadrature Mirror Filter

QoS Quality of Service

RAM Random-Access Memory

RFC Request For Comments

RIP Routing Information Protocol

RISC Reduced Instruction Set Computing

RMS Root Mean Square

ROM Read-Only Memory

RPE Regular Pulse Excitation

RPE-LTP Regular Pulse Excitation-Long Term Prediction

RTCP RTP Control Protocol

RTP Real-time Transport Protocol

RTT Round-Trip Time

RZ Return-to-zero

S-CSCF Serving-Call/Session Control Function

SB-ADPCM Sub-Band Adaptive Differential Pulse-Code Modulation

SCCP Skinny Call Control Protocol

SCS Service Capability Server

SCTP Stream Control Transmission Protocol

SDK Software Development Kit

SDP Service Delivery Platform

SFU Special Functional Unit

SGW Signaling Gateway

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SIP Session Initiation Protocol

SISD Single Instruction, Single Data

SLF Subscriber Location Function

SLI Scan-Line Interleave

SM Streaming Multiprocessor

SMP Symmetric Multiprocessor

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Management Protocol

SoC System on Chip

SP Stream Processor

SPE Synergistic Processing Elements

SPMD Single Program, Multiple Data

SRIO Serial RapidIO

SRTP Secure Real-time Transport Protocol

SS7 Signaling System # 7

SSE Streaming SIMD Extensions

SSH Secure Shell

SSL Secure Sockets Layer

STUN Session Traversal Utilities for NAT

SVC Scalable Video Coding

TCP Transmission Control Protocol

TCX Transform Coded Excitation

TDM Time-Division Multiplexing

TETRA Terrestrial Trunked Radio System

TISPAN Telecoms & Internet converged Services & Protocols for Advanced Networks

TLS Transport Layer Security

TUP Telephony User Part

TURN Traversal Using Relay NAT

UA User Agent

UAC User Agent Client

UAS User Agent Server

UDP User Datagram Protocol

UE User Equipment

UMA Uniform Memory Access

UMTS Universal Mobile Telecommunications System

UP User Part

UPSF User Profile Server Function

URI Uniform Resource Identifier

USB Universal Serial Bus

VAD Voice Activity Detection

VBR Variable Bitrate

VHDL Very High Speed Integrated Circuit Hardware Description Language

VLIW Very Long Instruction Word

VoATM Voice over Asynchronous Transfer Mode

VoIP Voice over Internet Protocol

VoN Voice Over Network

VoP Voice Over Packet

VQ Vector Quantization

VSELP Vector Sum Excited Linear Prediction

VToA Voice and Telephony Over ATM

WCDMA Wideband Code Division Multiple Access

WiMAX Worldwide Interoperability for Microwave Access

WLAN Wireless Local Area Network

XML Extensible Markup Language

Chapter 1

Introduction

We are currently living the convergence between fixed and cellular networks such as GSM, GPRS, CDMA and UMTS, and the Internet. This convergence, through Next-Generation Networks (NGNs), allows the same instances of multimedia services to be available for all the aforementioned networks.

Media Gateways are responsible for the translation of media formats, e.g. speech, among the different networks, in order to ensure the seamless interworking between them.

1.1 Goals and Contributions

The challenge of this dissertation is to create a software Media Gateway based on Asterisk, running on a commodity personal computer, and to take advantage of the processing capabilities of the now ubiquitous Graphics Processing Unit (GPU) in order to improve its performance, without any expensive custom-made hardware. The GPU part will focus mostly on audio transcoding.

1.2 Outline

This chapter introduces the motivation leading to this dissertation, and what contributions it will bring. Then, an overview of VoIP and the IMS architectural framework, will be made. Asterisk, the widely used open source telephony software will be reviewed. The basics of parallel computing will also be evaluated, in anticipation to the GPGPU discussion in Chapter 3.

Chapter 2 presents speech compression theory, along with several voice codecs currently in use, in order to expose their commonalities, and the reasons why they are usually implemented in DSPs.

Chapter 3 presents the GPGPU parallel computing implementation, focusing on NVIDIA's CUDA, as it is currently the most widely used GPGPU architecture. OpenCL is also briefly discussed, as both APIs will be used for an implementation.

Chapter 4 builds upon previous chapters and existing literature, in order to present directions to implement a Asterisk-based IMS PSTN/CS Gateway using a GPU, and a few tests debunking the GPU suitability for speech coder implementations.

Chapter 5 provides conclusions and suggests some other interesting topics related to Media Gateways that are worth exploring.

1.3 Telephony Standards

Public Switched Telephone Network (PSTN) is the network of the world's public circuit-switched telephone networks. It used to be analog, but currently it is mostly digital, except many last-mile installations (between a central office and a subscriber). It consists of analog telephone lines, fiberoptic cables (between offices, and on large customer installations), microwave transmission links, cellular networks, satellites and undersea telephone cables, and also includes mobile telephones. Any telephone in the world can connect to any other. Each telephone is identified by a number, which may include country codes and area codes. The numbering standards for the PSTN are the ITU-T E.163 and E.164. PSTN utilises standards created by the ITU-T. G.711, also known as "PCM for voice frequencies" is the speech codec used in PSTN [1].

Analog lines, also called **Plain Old Telephone Service (POTS)**, support only one call at a time. Analog lines have two matching interfaces: FXS and FXO. POTS lines terminate in a **Foreign eXchange Office (FXO)** device, such as a telephone handset, an analog modem, or a fax machine. The FXO device connects to an FXS (telephone jack on a wall). **Foreign eXchange Station (FXS)** supplies the dial tone and power. When a user connects a device (fax or phone) to the wall socket, he is connecting it to an FXS.

Telephone company customers connect their phones to the FXS ports provided by the company. In a situation where a Private Branch Exchange (PBX) is used, it must have both FXO and FXS ports. The FXO ports are used to connect to the FXS ports provided by the company, and the FXS ports are used to connect the phones or fax devices behind the PBX.

In telephony literature, **Digital Signal 0 (DS-0)** refers to the basic digital channel carrying a single phone call using 64 kbit/s, 8,000 samples per second at 8 bits each, using μ-law in North America and Japan, or A-law in Europe and the rest of the world. It is seen as the fundamental building block of digital telecommunication circuits.

The **E-carriers**, **J-carriers**, and **T-carriers** are all digital circuits based on DS-0s. For example, the T1 carrier is equivalent to 24 DS-0s, and support a bit rate of 1.544 Mbps. There are also **OC-carriers**, an evolution of the T-carrier, based on fiber optics technology. For example, the OC-1 carrier can carry the equivalent to 672 DS-0s. More advanced carriers, such as the OC-192 can carry the equivalent to 129,024 DS-0s. At densities above T3, OC circuits may be used [2].

Integrated Services Digital Network (ISDN) is a digital voice and data network, initially specified in 1988. It separates the bearer (traffic) and the signaling channels [3]. ISDN is available as:

- **Basic Rate Interface (BRI)**. It has 24 channels: one 16 kbit/s D channel and two 64 kbit/s B channels, for a total of two simultaneous calls. It is implemented mostly in Europe, and used in small installations.

- **Primary Rate Interface (PRI)**. It has 24 channels in the United States and Canada (one D channel and 23 B channels, for a total of 23 simultaneous calls), while in Europe it has 32 channels (one D channel, 30 B channels, and one synchronization channel that cannot be used). They are delivered over T1 lines in the United States and Canada, and E1 in Europe. Each channel operates at 64 kbit/s. It is very popular in the United States for larger installations.

The D channels stand for Data and transport out of band signaling (Q.931), enabling

features such as custom caller ID, and redirection of calls. The B channels stand for Bearing and transport payload [3].

Signaling System # 7 (SS7), is a suite of control protocols. There are two types of signaling: signaling between a subscriber and the local exchange (subscriber signaling), and signaling between exchanges (network, or trunk signaling). SS7 operates on the second category of signaling [4].

SS7 uses a separate packet-switched network to transfer signaling information for its two modes: circuit (voice) related, and non-circuit related services. Circuit related signaling provides setup and teardown of voice connections, while non-circuit related services are related to network management, number translation, and subscriber information retrieval. The SS7 stack consists of four layers. The lowest three layers are called Message Transfer Part (MTP), with functionalities such as electrical and optical transmission, reliable message transfer, and message routing and network management. The upper layer, User Part (UP), is divided into three types: Telephony User Part (TUP), which supports basic call setup and teardown on analog circuits; Data User Part (DUP), providing call control for circuit-switched services; and ISDN User Part (ISUP), a signaling protocol that is used to establish, maintain, and release circuit-switched connections. TUP and DUP are being replaced by ISUP [4].

1.4 VoIP

Voice over Internet Protocol (VoIP) is a set of protocols for the delivery of multimedia sessions over the pervasive IP, initially explored in the 1990s. It finds its uses mainly in telephony, where it may provide users with services beyond simple calls, such as voice mail, conference calls, Interactive Voice Response (IVR), caller ID, or call forwarding. VoIP implementations usually allow security features, such as encryption and authentication. Users can utilize VoIP through dedicated Ethernet or Wi-Fi telephones, analog telephone adapters, or softphones, software applications that use a computer's microphone and speakers.

IP networks are much less reliable than circuit-switched networks, working on a best-effort basis, with no well-established QoS guarantees, and this has an impact on the quality of the conversation. The ITU-T recommendation G.114 [5] limits the RTT to 300 ms, and when this value is exceeded, conversations tend to get confusing. Problems such as jitter may also appear, but VoIP implementations usually have mechanisms to mitigate them. As the available bandwidth is limited, voice has to be compressed, by employing the so-called speech coders.

H.323 and SIP are two of the most used VoIP stacks, while the well-known Skype VoIP application uses a proprietary, peer-to-peer protocol.

1.4.1 H.323

H.323 [6], first published in 1996, is an early recommendation from ITU-T that defines protocols and codecs for point-to-point and multi-point conferences over a packet network. It uses ITU-T and IETF protocols, as well as codecs defined by ITU-T and others. Figure 1.1 shows the H.323 protocol stack.

H.323 states that support for G.711 A-law and μ-law transmission and reception is mandatory for all terminals. Support for G.729 (and G.729a), G.723.1, G.726, G.722, G.728 and Speex is optional. Call signaling in H.323 is based on the H.255.0 protocol.

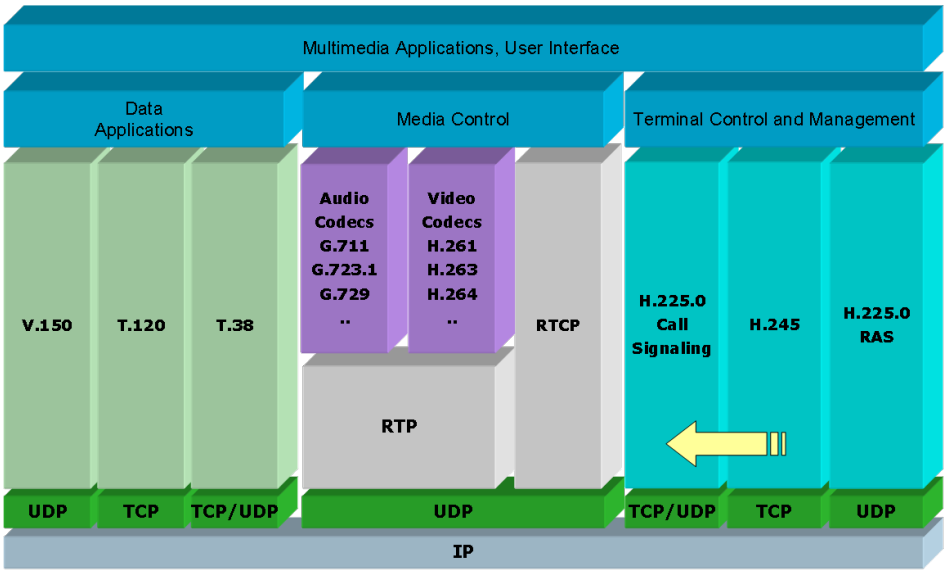


Figure 1.1: Complete H.323 protocol stack (from [7])

H.323 also specifies protocols for security and firewall traversal. Media is transported with RTP/RTCP or SRTP. There are two popular H.323 applications: Microsoft NetMeeting, and GnuGK [8].

1.4.2 SIP

Session Initiation Protocol (SIP) [9] is a text-based, human-readable signaling protocol developed in the late 1990s, defined by the IETF, and is used for controlling multimedia sessions such as audio and video over IP, but also instant messaging, presence information, file transfer and online games. SIP is the Internet equivalent to PSTN's SS7.

SIP is an Application Layer protocol and is independent of the Transport Layer, as it works with TCP, UDP, or SCTP. It borrows many of its elements from HTTP and SMTP, both IETF protocols. The request/response model and the error codes are similar to those of HTTP, and the addressing scheme, SIP URIs, is very similar to e-mail, in the form `sip:username@host` [10]. In addition, the E.164 PSTN number translation scheme is available, via ENUM.

Although SIP is only the signaling protocol in a VoIP stack, it is used interchangeably with the whole VoIP protocol spectrum. The protocols used in conjunction with SIP are:

- **Transport protocols**. TCP, UDP, SCTP, TLS.
- **Media transport and control protocols.** RTP, RTCP, SRTP.
- **Session negotiation.** SDP.

A myriad of services are built on top of SIP, such as call redirect and forwarding, caller ID, conference calls, instant messaging, and presence. In SIP, security is accomplished by the TLS and IPsec protocols, and NAT traversal is possible by using the STUN, TURN, and ICE protocols.

SIP has a number of server and client elements. The SIP client elements are:

- **User Agent (UA).** A SIP UA is a logical network entity at VoIP endpoints that can create and receive SIP messages in order to manage a SIP session. Among other requests, UAs can send registrations and invitations to sessions. A SIP UA is composed of a **User Agent Client (UAC)** and a **User Agent Server (UAS)**. The UAC sends SIP requests, and the UAS receives the requests and returns a response. A UAC and UAS exist only for the duration of a SIP transaction. The UA is usually present on the end user's device.
- **Back-to-Back User Agent (B2BUA).** Receives requests like a UAS, and after processing one, forwards the processed request on, as a UAC. The B2BUA retains state between calls. A stateful Proxy Server may contain a B2BUA.

The SIP server elements are:

- **Redirect Server.** Receives requests from a client, and replies with a new address/route path to the recipient. Especially useful in mobility.
- **Proxy Server.** The proxy server is an intermediary element that acts as both a UAS and a UAC, with the intent of making requests on the behalf of other clients. It routes the SIP request towards the next hop to the destination(s). It can also be used for authentication and accounting.
- **Registrar Server.** Users register their SIP UAs with this entity. It maps the users addresses to their SIP UA addresses.
- **Application Server.** Application servers provide an environment for delivering SIP-based multimedia services.

1.5 3GPP IMS

3GPP IP Multimedia Subsystem (IMS) is a standardized architecture for providing generic services, regardless of the underlying transport technologies. It provides a **Service Delivery Platform (SDP)**, accomplished via Application Servers, as well as reusable enablers and resources.

IMS has four main objectives: (1) combine the most recent technological tendencies; (2) attain the mobile Internet paradigm; (3) provide a common platform for the creation of multimedia services; and (4) increase revenue due to the expanding use of mobile networks [11].

IMS is seen by operators as a strategy for them to be more than simple "bit-pipes". By creating interesting services and by being involved in service delivery for third parties, instead of only supplying simple telecommunication features, their average revenue per user is expected to increase [12, 13].

Another objective of IMS is to facilitate network management, since everything is integrated in IP. It separates control and bearing functions, by having a service delivery network over a packet-switched infrastructure. IMS, being an access-agnostic platform, is fully compatible with the legacy infrastructure. The "walled-garden" business models typically followed by the network operators, in the sense that the operators don't reveal any data regarding their infrastructure design, is also reduced in IMS, which is an open standard. In IMS, there is a common user profile, with authentication, authorization, and accounting.

The IMS architecture was initially specified by the 3G.IP forum, and brought to the 3GPP as part of the UMTS specification, appearing first as 3GPP Release 5. ETSI TISPAN and 3GPP2 contributed to the standard [13]. The current specification is 3GPP Release 10 [14].

Many operators are adopting IMS as their infrastructure, allowing them to use it as an SDP. Existing services have to be migrated, either by rewriting them from scratch or by adding a compatibility layer. IMS also intends to supersede the **Intelligent Networks (IN)** architecture, used currently in both fixed and mobile networks, that allows the creation of telecom-based services. With IMS it is possible to quickly deploy services that are created once, and run anywhere, independently of the access [1, 15]. Push-to-talk, video sharing, mobile gaming, call centers, or Instant Messaging are examples of services that can be made on top of the IMS SDP. Content delivery such as news, music, infotainment (e.g.: horoscopes), games, or e-learning is also possible [12, 16, 17].

IETF protocols are used as much as possible, to facilitate the integration with the Internet, and to reduce any adoption barriers [17]. In order to manage multimedia sessions, IMS uses an extended version of the SIP protocol.

IMS exploits the convergence of services, access networks, and terminals. An IMS-based service can be used on many access technologies, whether it is fixed, mobile, or wireless access [1, 18].

IMS faces competition from Internet-based services, since a great percentage of terminals now have access to broadband data connections.

The core IMS architecture is formed by three layers [18], shown in Figures 1.2 and 1.3:

- **Transport/Access Layer.** Translates bearer and signaling channels between packet- and circuit-switched networks. Everything is translated to SIP/RTP, through media and signaling gateways. The objective is to allow the access to the IMS network on as many devices and user equipments with different access technologies (fixed, mobile, wireless) as possible. The media can be voice, video, or data. This layer allows IMS devices to receive and place calls to and from the PSTN or other circuit-switched networks, via the PSTN/CS Gateway. There can also be media servers, with functionality such as advertising and conferencing.

- **IMS/Control Layer.** This core layer comprises the CSCF SIP signaling servers, and the HSS.

- **Service/Application Layer.** The layers mentioned above yield a standardized, access-independent platform for the provision of multimedia services. This task is accomplished by the one or more application servers that reside on this layer, and these communicate via SIP with the Control Layer.

FOKUS developed an open source implementation of IMS core, Open IMS Core [20], and it serves as a reference implementation of it. It is not intended to be used in a commercial environment, but only for research purposes, typically on IMS testbeds. Open IMS Core implements the CSCFs and a lightweight HSS.

With the interconnection of IMS networks, the concepts of home and visited networks emerge. The infrastructure the user regularly connects to, is called the **home network**. During a roaming scenario, the operator may have an agreement with other operators, the **visited networks**, allowing the user to place and receive calls or use services there. The entity the user connects to in the home and visited networks is called P-CSCF.

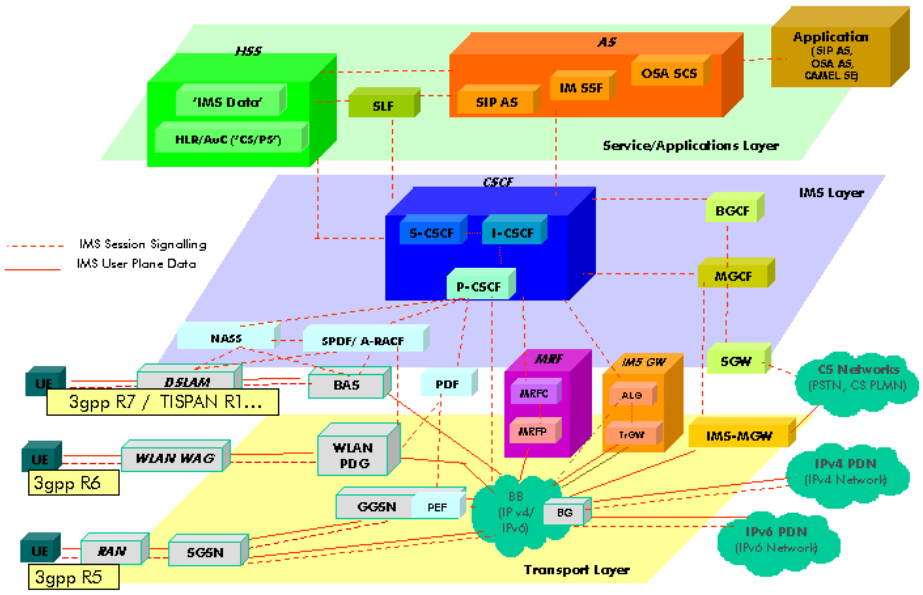


Figure 1.2: 3GPP/TISPAN IMS architectural overview (from [19])

In IMS, a user is identified by his **IP Multimedia Public Identity (IMPU)** and **IP Multimedia Private Identity (IMPI)**. Both are alphanumeric URIs, like `tel:+123-456` or `sip:johndoe@domain.com`. Each phone has an IMPI, and there can be multiple IMPU associated with an IMPI. IMPU is the public identifier that can be shared with other users.

3GPP standardizes functions, but not nodes (physical entities). Although it is usual to have one function per node, the implementers are free to combine functions into a single node, or split functions to multiple nodes [1].

1.5.1 HSS and SLF

The **Home Subscriber Server (HSS)** is an entity consisting of a user database that manages location information, security, authentication and authorization, user profile information (including the services the user is subscribed to), and the S-CSCF associated with the user. This information is used by network elements when managing sessions. It provides the subscriber's information such as his IP address to other IMS entities. HSS is an evolution of GSM's HLR and AuC. When multiple HSSs are used, another entity, called **Subscriber Location Function (SLF)**, maps user addresses to HSSs. HSS and SLF implement the Diameter protocol [1, 17, 11].

1.5.2 CSCF

Call/Session Control Function (CSCF) is a collection of SIP servers or proxies that manage SIP signaling within an IMS realm. It is able to setup, destroy, and route SIP sessions [11]. The CSCF is further divided into three sub-functions: P-CSCF, S-CSCF, and I-CSCF.

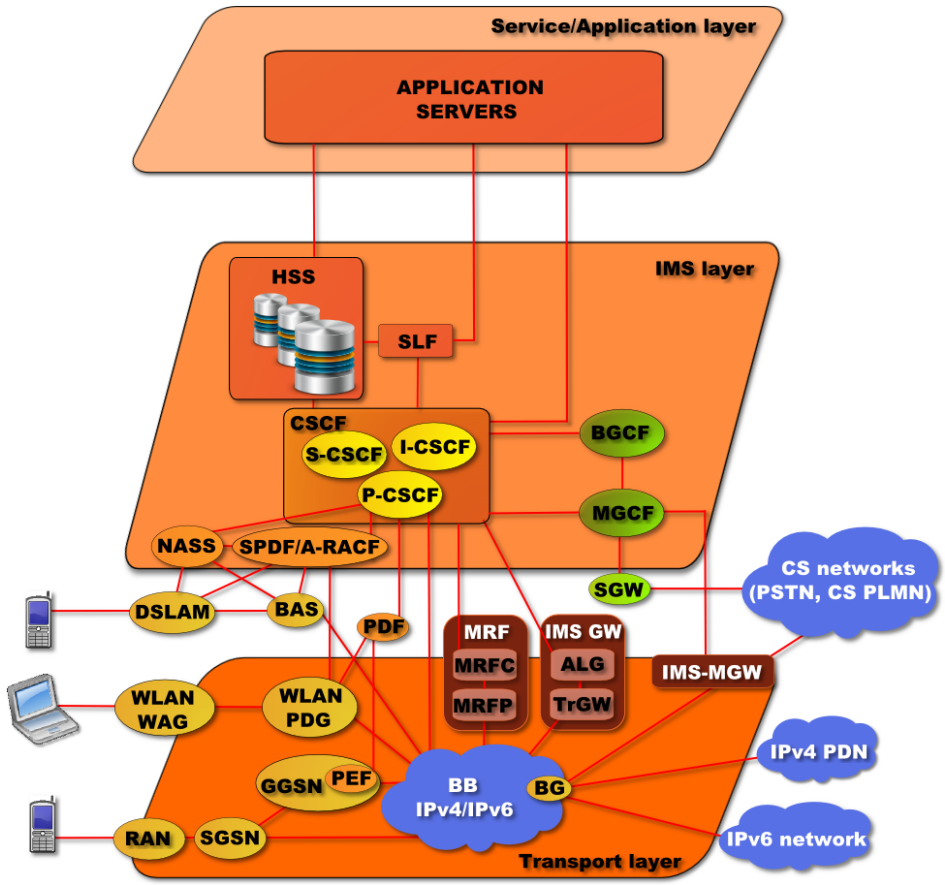


Figure 1.3: 3GPP/TISPAN IMS architectural overview - HSS in IMS layer (as by standard) (from [19])

A **Proxy-Call/Session Control Function (P-CSCF)** is a SIP proxy server, and represents the IMS contact point for the user's SIP signaling. Every single message from/to a IMS terminal is first routed through its P-CSCF. An IMS terminal is assigned a P-CSCF before registration. Authentication is made with the P-CSCF, and this information is passed to other nodes in the IMS network, so that authentication is needed just once. The P-CSCF has also other functions like generating charging information, SIP message compression, and QoS management. This entity is either located in the visited network, or in the home network, if the visited network IMS is not IMS-compliant. For scalability and redundancy reasons, there are usually multiple P-CSCFs in an IMS network [1, 11].

The **Serving-Call/Session Control Function (S-CSCF)**, the central node of the signaling plane, is a SIP proxy with a few added functionalities. It routes the SIP request to the destination, as SIP signaling always traverses the P-CSCF and the S-CSCF. It also performs session control, and acts as a SIP registrar, meaning it maps the user location (IP) and the user's SIP address (IMPU). There is a Diameter interface to the HSS. The HSS provides, through this interface, registration, authentication and user profile information, so that the I-CSCF can assign an S-CSCF to the user. A set of triggers, called **user service profile**, is defined in the user profile, that can result in a SIP message to be routed through one or more ASs, for service provision. All SIP messages from the UEs pass through their

allocated S-CSCF. In a situation where the user dials a telephone number instead of a SIP URI, the S-CSCF translates it using the E.164 standard (the regular numbering plan), and performs a breakout to the PSTN, by selecting an appropriate BGCF. Due to scalability and redundancy reasons, an IMS network might use more than one S-CSCF. The S-CSCF is located in the home network [18, 1, 11].

Finally, the **Interrogating-Call/Session Control Function (I-CSCF)** is a SIP proxy that sits on the edge of an administrative domain. The I-CSCF queries the HSS through a Diameter interface, retrieving the user location, and routes the SIP message to the assigned S-CSCF in the destination realm. The address of the I-CSCF is listed in the domain's DNS records, so that it can be the next SIP hop on a given IMS realm. The I-CSCF is located in the home network, or in a visited network in some situations, and there can be multiple I-CSCFs in a network [1, 11].

1.5.3 Application Server

As mentioned before, one of IMS' aims is to deliver multimedia services, like voice mail or push-to-talk. The **Application Server (AS)** is a SIP entity that hosts and executes those multimedia services. It can be a SIP proxy, a SIP UA, or a SIP B2BUA, depending on the service requirements [1].

The native IMS AS is the SIP AS. However, IMS can interface with legacy application servers such as those from CAMEL via IM-SSF, and OSA framework via OSA SCS compatibility layers, to ensure the existing services remain functional. These, like the native ASs, are regarded as SIP proxy servers, SIP UAs, or SIP B2BUAs. If there arises a need to process multimedia, the AS calls the appropriate MRFC/MRFP [1].

The AS interfaces with the S-CSCF via SIP. The latter, based on a list of rules, might redirect SIP messages to the AS for the provision of services, and transfer accounting information to the charging module.

An AS can be located either in the home network, or in an external network. If the AS is in the external network, there is no interface with the HSS. By allowing external application servers, this facilitates third party service development and billing-related matters, while keeping control of the core services [1, 18].

1.5.4 MRF

The **Media Resource Function (MRF)** is a media server used to supply audio, video, and text streaming. MRF can be decomposed into two nodes: the **Media Resource Function Controller (MRFC)**, and the **Media Resource Function Processor (MRFP)**. The MRFC is responsible for managing the MRFP via a H.248/Megaco interface, and interacts with the ASs using the SIP protocol.

Multimedia processing features often required by the ASs, like multimedia advertisements, audio and video conferencing, transcoding, echo cancellation, DTMF generation/detection, voice mail, and voice recording and processing, are supplied by the MRFP [18].

1.5.5 BGCF

Breakout Gateway Control Function (BGCF) is a SIP proxy server responsible for routing based on telephone numbers, when an IMS terminal is performing a call to a circuit-switched network.

When the IMS session is forwarded into the PSTN/CS domain, the BGCF has to select an MGCF. The breakout can occur in the same network, in which case the BGCF selects an MGCF in the same network, or it will forward the invite information flow to the BGCF in another network. The BGCF handles requests from an S-CSCF, when the latter determines that it is impossible to route the session with DNS or ENUM/DNS. For a PSTN call, the breakout is performed when the E.164 address cannot be resolved into a SIP URI [1, 21].

1.5.6 PSTN/CS Gateway

The existence of a device that interfaces with the PSTN is of the utmost importance in IMS [11]. The objective is to make an interface between the packet-based IMS network and the PSTN/CS networks, making receiving and placing calls (or other multimedia sessions) from and to any CS network possible, as well as roaming of IMS subscribers [21]. This is accomplished in IMS through the **PSTN/CS Gateway**.

PSTN networks use ISUP (or BICC) over MTP (an SS7 protocol), while IMS uses SIP for signaling. For media, PSTN networks use PCM, while IMS uses any media supported by RTP. There is a need for conversion between both [18, 21].

The PSTN/CS Gateway receives requests from the BGCF and interfaces with the CS networks. This gateway is divided into three nodes: an MGCF, an SGW, and an MGW.

Figure 1.4 shows the nodes that form the PSTN/CS Gateway, assuming the existence of an SS7 PSTN network. The BGCF is not a part of the PSTN/CS Gateway.

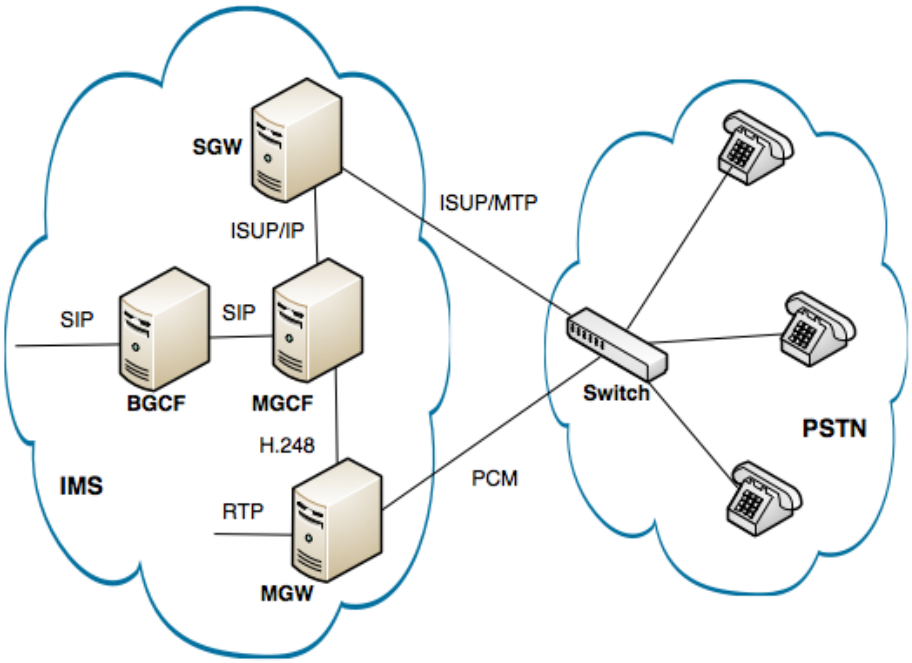


Figure 1.4: The PSTN/CS gateway interfacing a CS network

SGW

A **Signaling Gateway (SGW)** interfaces with the signaling plane of the CS network. It performs lower layer protocol conversion between the IP and CS networks. For example, it transforms SCTP (an IP protocol) into MTP (an SS7 protocol) to pass ISUP from the MGCF to the CS network [1].

MGCF

Media Gateway Control Function (MGCF), the central node of the PSTN/CS gateway, controlled by the BGCF, is a SIP endpoint that performs call control conversion by mapping SIP (on the IMS side) to ISUP/BICC (protocols in the CS networks) over IP. Additionally to the call control protocol conversion, in order to place or receive calls to/from the CS network, the MGCF controls the resources in a Media Gateway, using an H.248/Megaco interface [1, 21].

MGW

The **Media Gateway (MGW)**, sometimes written with the MG acronym, the main topic of this dissertation, serves as an interface to the PSTN/CS network media plane, and is controlled by an MGCF.

A non-IMS Media Gateway is a translation device or service that converts digital media streams (audio, video, or text) between disparate telecommunications networks such as circuit-switched networks PSTN (DS-0), SS7, Radio Access Network (2G, 2.5G, 3G) or PBX, and media streams from a packet network (e.g., RTP). MGWs are expected to be highly reliable devices and be able to handle up to several thousands of digital circuits. They are also expected to support several signaling and call control protocols; a wide range of codecs; DTMF detection and generation, echo cancellation; conference calls; and often the basic routing and management protocols [22].

Transcoding, or the conversion between speech formats, is needed when two entities do not have matching speech codecs. As an example, the IMS terminal might be using AMR, but the PSTN terminal might be using the G.711 codec. With rare exceptions, the traditional method of transcoding between two different formats is made by decoding the incoming bitstream encoded in one format, and re-encoding it in another format, a task known as **tandem transcoding**. The computationally intensive transcoding operation will be studied in detail.

On the IMS side it sends and receives media over RTP. On the PSTN/CS side it connects to the CS network via one or more PCM time slots (G.711) in TDM. It performs the conversion of bearer traffic between these two networks, but also payload processing like transcoding, echo canceling, or conference bridging [1, 21].

There are four signaling protocols used by MGCs to control non-IMS Media Gateways: H.323 (via the H.225.0 and H.245 protocols), SIP, MGCP, and H.248/Megaco. However, in IMS, only H.248/Megaco is required, hence the MGCF controls its Media Gateway via the H.248/Megaco protocol [1].

Hardware MGWs and MGCs are typically manufactured by industry leaders in networking hardware, such as Cisco [23], Nortel [24], Avaya [25], Dialogic [26], Squire Technologies [27], Metaswitch Networks [28], and TelcoBridges [29].

The interfaces that are often supported include PRI, GSM, Ethernet, T-carrier, and Optical carrier. The currently sold Media Gateways usually support the basic speech coders G.711, G.722, G.726, G.729, and GSM-FR, and on a lesser extent, codecs like G.722.1, G.722.2 (AMR-WB), G.723.1, G.729A/B, AMR, GSM-EFR, iLBC, ADPCM, LPC-10, Speex. On carrier-grade systems, routing protocols (BGP, IS-IS, OSPF, RIP, etc.) are often supported, as well as management interfaces (SNMP, CLI, Telnet, or SSH).

Software Media Gateways can be made with telephony software such as Asterisk [30] and FreeSWITCH [31], in cooperation with telephony interface cards. On an IMS context, some attempts have been made with relative success. These will be reviewed later in Chapter 4, during the discussion of the implementation.

1.6 Asterisk

Asterisk [30] is a highly customizable, open source telephony engine and toolkit software. It is dual licensed, under both GNU GPL and a proprietary license. It runs on Linux, Mac OS X, OpenBSD, FreeBSD, and Solaris UNIX-like operating systems.

Asterisk was originally created by Mark Spencer, after realizing the existing PBX solutions were too expensive for his small company. The first release of Asterisk, version 0.1.0, happened on December 5, 1999. The first major release, 1.0, was released on September 23, 2004 at the Astricon, the official Asterisk user and developer's conference. Throughout the years, Asterisk gained support for many VoIP protocols and Asterisk server interconnection, making it one of the most popular software-based VoIP PBXs. Behind Asterisk is a company called Digium, which sells hardware for Asterisk and certifications [32].

Many companies and universities are replacing thousands-phone strong installations with Asterisk, due to licensing costs and security concerns. Telephone companies are also using Asterisk to handle VoIP. Asterisk can augment, or entirely replace an existing telephone system, whether the user is a hobbyist with a single telephone line, or an executive running a large call center with multiple PRIs [32].

Thanks to its modularity, Asterisk supports a wide range of codecs and VoIP protocols [33, 34].

1.6.1 Applications

Asterisk, with its wide support for VoIP protocols, interoperability with many existing telephony standards, and modularity, can act as the following [33]:

- **Private Branch Exchange (PBX).** Also called a **Switch**, the PBX is essentially a private telephone network for use within an organization, and must be able to handle many extensions. With an Asterisk PBX, there's the added functionality of packet network and circuit switched network bridging. Also it is fully programmable and customizable, so any desired functionality is possible, meeting the demands of different businesses. Asterisk allows users to replace existing expensive PBX solutions, sometimes even allowing more functionality. Asterisk spans the hobbyist with only one telephone line, to businesses running large call centers with multiple PRIs, allowing the creation of PBX systems that rival the traditional PBXs in terms of functionality, while being cost effective and with low maintenance.

- **MGW/VoIP Gateway.** Asterisk can serve as the basis of a Media Gateway, by interfacing the PSTN and VoIP networks, when coupled with telephony interface hardware. Due to Asterisk's modularity, it is possible to support a myriad of protocols and codecs. If necessary, transcoding between codecs is done. Note that Asterisk, out-of-the-box, is not IMS-compliant. This issue will be addressed later.

- **Conference Bridge.** Asterisk comes with a conference calling system, called "MeetMe", with features such as PIN protection, recording, and Music on hold [32].

- **Interactive Voice Response (IVR)/Call Center.** Asterisk can be used to make **Automatic Call Director (ACD)** systems, without requiring physical presence, allowing the employee's own broadband connection to be used with VoIP or PRI [2]. The ACD routes the calls, and advanced scenarios such as skills-based routing, are possible. Also useful for call centers are **call queues**. These are employed when there are a number of "answerers" answering calls from a much larger set of "callers", such as a customer service department [32].

- **Voicemail.** Including features such as voicemail directory, forwarding, and playing of messages depending on the situation [32].

1.6.2 Architecture

Asterisk is designed to be flexible. A set of APIs is built around a central PBX core, abstracted from the protocols, codecs, and hardware interfaces that telephony applications might use. The PBX core connects calls between various users, and performs automated tasks. This way, Asterisk is future-proof, allowing any available technologies to be integrated. There are four APIs defined for loadable modules on Asterisk: Channel API, to handle a type of connection, e.g. SIP, PRI, or other technology; Application API, so that applications like voicemail or conferencing can be made; Codec Translator API, to provide encoding and decoding support for several audio formats; and File Format API, in order for Asterisk to be able to read and write several file formats. New modules can be developed using these C APIs outside of the core functions [33]. Using these APIs, Asterisk has modules for such disparate purposes as Music on hold, speech recognition or database support.

There are three module managers in Asterisk [33]: Application Launcher, responsible for launching applications that perform services such as voicemail, file playback, and directory listing; Codec Translator, that uses codec modules for the encoding and decoding of various audio compression formats used in the telephony industry; and the Scheduler and I/O Manager, responsible for handling low-level tasks and system management.

Figure 1.5 shows an architectural overview of Asterisk.

Asterisk is a heavily multi-threaded application. The most important types of threads for this dissertation are the Channel threads (also called PBX threads), and the Network Monitor threads (they monitor for incoming requests). An abstraction over POSIX Threads (pthreads) is used to manage threads and locking [35].

1.6.3 Channels

According to the Asterisk documentation [30], a "phone call through Asterisk consists of an incoming connection and an outbound connection. Each call comes in through a channel

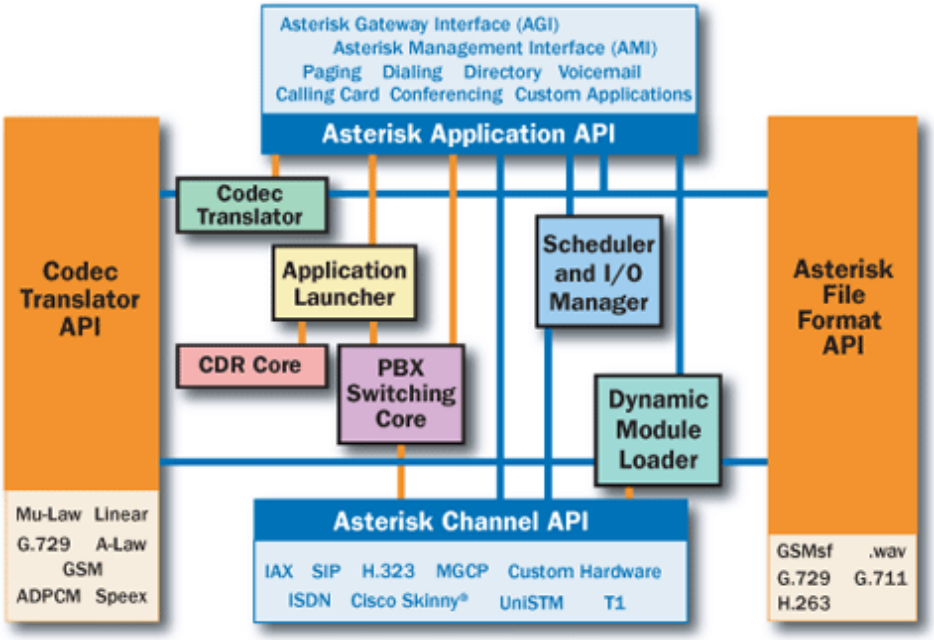
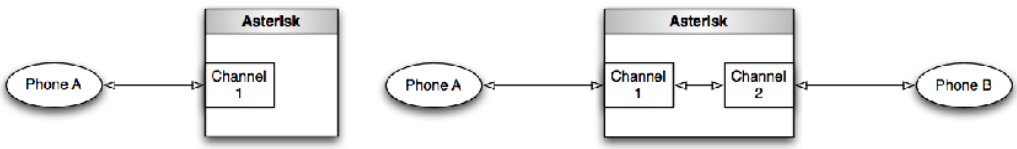


Figure 1.5: Asterisk Architectural Overview (from berklix.org)

driver that supports one technology, like SIP, DAHDI, IAX2, etc.". This means that a simple call is made of two **channels**, one inbound and one outbound. A channel can also be seen as a connection between Asterisk and an endpoint. When there is an incoming call on a channel driver interface, the interface creates an internal PBX channel, starts a PBX thread on the channel (**channel thread**), and the channel runs a dialplan. The **dialplan** tells Asterisk how to handle inbound and outbound calls, and consists of a series of instructions. If there is a Dial application in the dialplan, a second channel is created for the outbound call, and the two channels are bridged (assuming the outbound channel is answered), allowing the audio to pass between each other, forming a call. In a scenario of a conference bridge ("MeetMe"), more channels are bridged in order to pass audio between each other. As an example, when a user is checking his voicemail, there is only one leg, and one channel [36, 37]. Figure 1.6 illustrates the definition of channels.



(a) One call leg, one channel (b) Two call legs, two channels

Figure 1.6: Relationship between legs and channels on Asterisk (from [36])

If during a call set up, the endpoints cannot agree on a common audio codec, transcoding is required. A new instance of an encoder and decoder is created on the PBX thread.

Asterisk systems typically support tens to hundreds of calls on machines with dual-core

processors. Of course, there is also a limitation imposed by the network interfaces. Important factors in performance are codec choices (e.g., G.729 is more CPU-intensive than G.711), use of echo cancellation, and use of conferencing [2, 38].

1.6.4 Hardware

Asterisk supports a myriad of telephony interface cards to interface with the PSTN/CS networks, as well as transcoding and echo cancellation cards, designed to offload the audio transcoding process from the CPU.

The current range of telephony interface cards currently in the market is:

- **Analog Cards.** Allows Asterisk to connect to analog systems (POTS), phones, or fax machines. These are good for small installations, where not many channels are required [3]. Each channel requires an FXO port, and these cards are usually bundled with 4, 8, or 24 FXO/FXS ports. They come in PCI form.

- **Digital Cards.** With these, cards Asterisk can connect to T1, E1, J1, and BRI (ISDN) interfaces. These are usually cost-effective when requiring more than 10 circuits [3]. Digium sells cards with 1, 2, and 4 T1/E1/J1 ports, as well as 4 S/T Ports. They are sold in PCI and PCIe forms. These cards are known to work with SS7 networks, with chan_ss7 [39] or Digium's libss7 libraries.

- **Hybrid Cards.** Using these cards, Asterisk can connect to both analog and ISDN networks. They feature both analog and digital BRI ports.

Telephony interface card manufacturers include: Digium [40], XORCOM [41], ATCOM [42], BroadTel [43], DigiVoice [44], voicetronix [45], Sirrix [46], Parabel [47], Nicherons [48], PhonicEQ [49], Rhino [50], Sangoma [51], Dialogic [26], Elgato [52], Pika [53], and Synway [54].

In order to assist the transcoding process, PCI and PCIe transcoding cards are sold, offloading this work from the CPU. All Digium cards [40] are able to encode/decode 120 G.726 channels or 92 G.723.1 channels, with a DSP. They also offer support for G.729a; and direct conversion between G.729a or G.723.1 into μ-law or A-law. When using a transcoding card, Asterisk uses a different codec module that interfaces directly with the transcoding card's driver.

There are also cards made with the purpose of performing echo cancellation in PSTN interfaces, for use with the telephony interface cards, as this process requires a great amount of processing power, replacing Asterisk's software echo cancellation [2].

Several manufacturers sell transcoding cards with support for many codecs, compatible with Asterisk and FreeSWITCH, for the VoIP and PBX market: Digium [40], OpenVox [55] Sangoma [51], Signalogic [56]. Third-parties transcoding cards typically support the G.711 (A-law and μ-law), G.722, G.722.1, and G.726 codecs; other codecs such as G.722.2 (AMR-WB), G.723.1, G.729A, G.729AB, AMR, GSM-FR, GSM-EFR are supported in a lesser extent. These cards usually employ DSPs [57, 58], and cost in the hundreds to thousands of dollars range [59, 60].

Asterisk uses the **Digium Asterisk Hardware Device Interface (DAHDI)** (formerly Zaptel) interface to control Digium's telephony interface and transcoding cards. Third-party transcoding cards (not manufactured by Digium) don't use the DAHDI mechanism. As an example, Sangoma cards are recognized as Ethernet adapters, and an Asterisk codec

module has to be installed, as well as a controller to allow card use by multiple Asterisk and FreeSWITCH instances. Sangoma cards also allow the transcoding to be done in a remote computer.

1.7 Parallel Computing

Although present in the academia and the super-computing community for some time, parallel computing only recently started to penetrate the consumer market with multi-core CPUs in computers, mobile phones and tablets, as well as programmable many-core graphics cards.

1.7.1 Motivation and Challenges

The CPU speed increased historically according to Moore's Law. This law states that the number of transistors of a CPU doubles every 18-24 months. Until the mid-2000s, one of the main sources of speed increase was frequency scaling. By increasing the frequency, the time it takes to execute an instruction decreases. From the beginning of the 2000s and until the mid-2000s, clock speeds escalated from about 1 GHz to 3.8 GHz. This frequency scaling, however, was not sustainable, as it has several physical consequences. The raise of the clock speeds, and at the same time the increase of transistor count and density, resulted in overheating and climbing power consumption. Also, memory access time could not be reduced at the same rate as the CPU clock period [61].

CPU manufacturers arrived to a solution, shipping shared memory multi-core architectures. Typically, the cores are integrated onto a single die (chip multiprocessor), or onto multiple dies in a single chip package.

Limited parallel computing has been mainstream on personal computers through the pipelined execution of instructions and the use of multiple functional units in CPUs, starting in the mid-1990s. CPUs feature vector processing extensions such as Intel SSE and MMX, or AMD 3DNow!, but these techniques have stagnated [62].

By 2012, the norm is dual- and quad-core computers, backed up by operating systems that are able to run applications in parallel.

The holy grail of parallel programming would be the automatic parallelization for a given architecture of a sequential program, but work with this objective has been around for more than two decades, without noteworthy results. It is then the programmer's responsibility to do this adaptation. There is a whole body of research in parallel programming languages and environments, with the objective of giving the programmer a comprehensive level of abstraction [62].

Parallel programming involves knowledge in the following areas [63]:

- **Computer architecture.** Memory organization; caching and locality; memory bandwidth; Flynn's Taxonomy and variants, floating-point precision versus accuracy.
- **Programming models and compilers.** Parallel execution models, types of available memories, array data layout and loop transformations, for the arrangement of data and loop structures.
- **Algorithm techniques.** Tiling, cutoff, binning, and their implications in scalability, efficiency, and memory bandwidth.

- **Domain knowledge.** Knowledge of the domain of application, and general numerical methods and other mathematical concepts.

1.7.2 Flynn's Taxonomy

Flynn's taxonomy classifies computer architectures according to their number of concurrent instruction (or control) and data streams [62, 64]:

- **Single Instruction, Single Data (SISD).** This architecture exploits absolutely no parallelism (instruction, nor data). This is the conventional von Neumann model, and the approach in older, uniprocessor computers, where there is only one processing element, and it has access to a single program and data storage. This processing element fetches an instruction and the associated data, and executes the instruction, saving the result in the data storage.

- **Single Instruction, Multiple Data (SIMD).** Exploits data parallelism, as a single instruction stream performs operations on multiple data streams, utilizing several processing units. As an example, a SIMD multiplication instruction could perform two or more multiplications on different sets of input operands in parallel [65]. Examples are GPUs (sometimes NVIDIA uses the term SIMT), vector (or array) processors such as the SSE and MMX processor extensions, or Cell.

- **Multiple Instruction, Single Data (MISD).** Multiple instruction streams operate on a single data stream. Used mainly for fault tolerance, when different machines must agree on a result, for example in the Aerospace industry. Each processing element has a private program memory, but there is only a single global data memory. Each processing element fetches the same data from the data memory and loads an instruction from the private program memory. The instructions are then executed in parallel.

- **Multiple Instruction, Multiple Data (MIMD).** Multiple processors operate on different data with different instructions. Multi-core machines and clusters are examples of MIMD systems. Some authors further divide MIMD in more categories: SPMD, and MPMD. In Single Program, Multiple Data (SPMD), multiple processors execute the same code (at independent points, versus in the lockstep on regular SIMD) on different data. This architecture can be seen in message passing, distributed memory architectures. In Multiple Program Multiple Data (MPMD), multiple processors execute at least two independent programs. The Cell processor is an example of this architecture.

1.7.3 Parallelism in Microprocessors

One of the speedup strategies in the early days of personal computing was to increase the processor's word size, or in other words, the amount of information the processor can handle in a cycle. This strategy is called **bit-level parallelism**, and it allowed the reduction of the number of instructions needed to perform an operation on variables with a size greater than the length of the word. This trend came to a halt with the advent of 32-bit processors, although current systems have up to 64-bit word sizes [62].

Programs are nothing but a set of instructions. Through **instruction-level parallelism**, or **pipelining**, these instructions can be combined into groups and overlapped, without changing the result of the program. Processors currently feature instruction pipelines with multiple

stages, where each stage corresponds to a certain action. A direct consequence is that an N-stage pipeline can be executing N instructions at different stages. The most basic pipeline of a RISC processor, has five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory access (MEM), and Write-Back (WB). This parallelism resembles an assembly line in an automobile factory, where there are dedicated units for a given task. The execution of the different stages should take the same amount of time, determining the cycle of the processor. In a no-dependency scenario, in each clock cycle the execution of one instruction is finished and the execution of another started [62].

In **parallelism by multiple functional units**, processors can have several Arithmetic Logic Units (ALUs), Floating-Point Units (FPUs), load/store units, or branch units. The units are able to execute in parallel, i.e., different instructions can be issued to the different units to be run in parallel. There are two types of multiple-issue processors: **superscalar processors**, which can fetch and dispatch multiple instructions at a time; and **Very Long Instruction Word (VLIW) processors**, in which the compiler determines independent instructions and encodes them in a single long instruction word [62].

The techniques mentioned so far assume a single sequential control flow provided by the compiler, determining the execution order between instructions. However, the degree of parallelism is very restricted for these techniques, and their ceiling has been reached for some time. The solution was to put multiple, independent processor cores onto a single microprocessor chip. Each of the cores has a separate flow of control, i.e., parallel programming techniques must be used [62].

1.7.4 Memory Organization

Parallel computers can be seen as a collection of processing elements connected through some network interface [66]. Parallel computers can be classified by their main memory organization as **shared memory** machines and **distributed memory** machines. Figure 1.7 schematizes the difference between these architectures. However, many parallel computers have a **hybrid distributed-shared memory** architecture, with multiple processors in each computer, connected by a network. Most supercomputers use this hybrid architecture, by having a shared memory part in the form of SMPs and/or GPUs, and a distributed memory part, by having multiple networked machines communicating with each other [67].

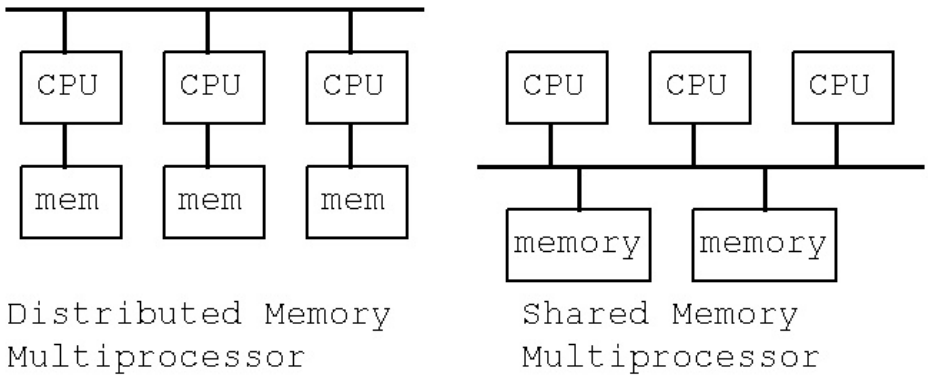


Figure 1.7: Difference between distributed and shared memory parallel computers

Shared Memory

In **shared memory**, a common data memory is shared between all processors in a single address space. Shared memory systems are usually **Uniform Memory Access (UMA)** systems, or systems where any element of the main memory can be accessed with the same latency and bandwidth. An example of a UMA system is the **Symmetric Multiprocessor (SMP)**, such as is the case of the current multi-core CPUs [64].

Shared memory computers do not scale very well due to the need of having **cache coherence**. If there are no mechanisms to ensure each processor's cache remains consistent, that is, when the same value is stored in more than one location it should be consistent, then there is a chance that the program will execute incorrectly. It is very hard to design fast cache coherence systems with the increase of the number of processors [64].

Typical APIs used in shared memory implementations are POSIX Threads, OpenMP and message passing. Multi-threading was historically used for functional separation purposes, but is now being increasingly used for parallel programming, by having one thread running on each physical processor at a given time. Semaphores and monitors are some programming mechanisms used for shared memory programming in order to guarantee data integrity on specific memory addresses [66].

Distributed Memory

In **distributed memory**, each independent processor has its own local address space (own memory), and communicates with other processors' memories for data transfer or synchronization via I/O (message passing) operations, through a communication network (shared memory bus, network, etc.). There is not a global address space for all the processors. This makes the memory scalable with the number of processors, as there is no cache coherence. Distributed systems are **Non-Uniform Memory Access (NUMA)** systems, where the memory elements can't be accessed with the same latency and bandwidth, due to being physically separated [64].

Cluster computing is an example of a distributed memory system where, although not mandatory, the set of computers that form a cluster are identical. The most relevant project is the Beowulf cluster [68]. **Grid computing** differs from cluster computing as the computing nodes are not required to be identical. On the same project there can be processors of different architectures, or even GPUs. The most relevant software is BOINC, a grid computing middleware. Projects such as SETI@home [69] or Folding@home [70] are now adopting this middleware.

APIs such as MPI are generally used to program distributed memory systems, and recent frameworks such as Apache Hadoop [71], an implementation of Google's MapReduce, perform distributed computing on large data sets using clusters of computers.

1.7.5 Quantification of Improvement

A parallel program can be decomposed into a sequential portion and a parallel portion, and the maximum speedup achieved by the program is determined by Amdahl's Law or Gustafson's Law [62].

Amdahl's Law, formulated in 1967, states the maximum expected improvement of a system when only a part of it is improved. It is especially important in parallel computing, where

it can predict the maximum speedup when using multiple processing units. The speedup of a program running in parallel is always limited by the sequential part of the program.

If P is the portion of a program than can be made parallel, and $(1 - P)$ is the portion that cannot be parallelized (remains serial), then the overall speedup that can be achieved by employing N processors is:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

As N tends to infinity, the **maximum speedup by Amdahl's Law** can be derived:

$$maximum\ speedup = \frac{1}{(1 - P)}$$

Through this law, one arrives to the conclusion that even if 90% of the code is parallelizable, only a $1/0.1 = 10$ times speedup is achievable. In other words, the performance improvement is bounded by the portion that cannot be run in parallel. For example, if a program needs 20 hours to complete on a single processor core, and only a portion of 1 hour cannot be parallelized, then, no matter how many processors are devoted to the parallel execution of the program, the minimum execution time cannot be under one hour, and the speed up is limited to 20x.

Gustafson's Law amended Amdahl's Law in 1988, by stating that the same amount of time can be used to solve a larger problem, while the Amdahl's Law assumes that the problem has a fixed size. With the development of new and more complex applications, the size of the problems tends to grow. The **maximum speedup by Gustafson's Law**, S, employing N processors, and having α as the non-parallelizable portion of the code is given by:

$$S(N) = N - \alpha(N - 1).$$

1.7.6 Stream Processing

Stream processing is a parallel programming paradigm related to SIMD, and recently being harnessed in the personal computer domain through the GPGPU paradigm.

Stream processing adopts a *gather*, *operate*, and *scatter* style of programming. The **gather** operation collects the data from sequential, strided, or random memory locations in an array into a **stream**, a set of data of the same type. Given a stream, a series of operations, **kernel functions**, are applied to each stream element. The operation of kernels resemble the loops in a serial program, where operations are performed upon each element of, for example, an array, in each loop. Finally, the **scatter** operation writes the result data into a memory location [72].

Stream processing is well-suited for graphics and image processing applications, which exhibit sequential load/stores. Applications that perform random accesses to memory, such as many scientific applications that involve irregular grid/mesh computations, are more challenging, as there is a need to have index arrays before gathering/scattering the data [72].

Research stream processor implementations include Stanford's Imagine [73] and Merrimac [74]. Commercial stream processors include IBM's Cell, that contains a set of SIMD coprocessors, SPEs, making it a MIMD processor; and GPUs mainly from NVIDIA and AMD.

CUDA, a GPGPU architecture made by NVIDIA, and OpenCL, an open standard for heterogeneous computing, will be reviewed in Chapter 3. DirectCompute, Microsoft's take

on GPGPU, and part of the DirectX 10 and 11 APIs will not be discussed. Obsolete GPGPU APIs will be mentioned as part of a historical review.

1.7.7 Parallel Algorithms

Parallel algorithms must be designed with as independent as possible work units. With less frequent work unit communication comes better performance and scalability.

Data parallelism, or **loop-level parallelism** is the ability to execute parts of the same task, with different data, at the same time, and it is usually visible in program loops, when there are no dependencies between iterations. These exhibit an intrinsically horizontally parallelizable nature. The same set of operations is performed on all elements, and different nodes can do it in parallel. A matrix multiplication can be easily parallelized, as each cell being calculated does not depend on other cells. On the other hand, the computation of a Fibonacci sequence cannot be parallelized, as each loop iteration depends on the output of previous iterations (loop-carried dependency). GPGPU, vector and SIMD processors exploit this type of parallelism [62].

Task parallelism (or **functional parallelism**) is the ability to execute different tasks of a given problem at the same time, and can be seen as the opposite of data parallelism. In data parallelism, the same instruction is performed on multiple data elements. In the case of task parallelism, different instructions are performed on the same or different data, in a pipelining manner. As an example, in a program with a GUI, there can be a task for updating a GUI, and other for reading a database. These tasks are unrelated, so multiple **threads** can be dealing with these different functionalities. **Synchronization** might be necessary when competing for resources. Multi-core superscalar general-purpose processors implement task parallelism [62].

Problems should be broken into a set of work items that can be given to multiple processors. The programmer has to analyze which parts are good candidates for parallelization, and which have to remain serial, either by **task decomposition**, identifying the algorithm's tasks and their dependencies so that each processor can work on a task; or **data decomposition**, identifying the chunks of a data set that can be processed in parallel, so that each processor can work on a portion of the data. Sometimes, the parts that are parallelizable can be easily determined, because they do not need any form of coordination or communication [63, 75].

Concurrent programming, due to its non-deterministic nature, introduces challenges and complexities such as dead-locks, race conditions, and synchronization. Current research is developing implicit parallelism, with the objective of relieving the programmer from these challenges, by building libraries, tools, and programming language constructs; but this kind of technology is not yet ready for prime time.

The programmer has to face the compromise among parallelism, computational efficiency, and memory bandwidth consumption. The process of **computational thinking**, or the act of decomposing problems in algorithms, has a great share of art in its essence, but a programmer with a strong knowledge of the parallel architectures, with their particularities, bottlenecks, and tradeoffs, has a great advantage when choosing algorithms. Some algorithms require fewer steps than others in order to perform a computation; some allow a higher degree of parallelization than others; and some require less memory bandwidth than others. Usually the programmer has to make the best compromise possible between these three variables for a given hardware system, as there is often not an algorithm that performs better than all the others in all aspects [63].

Factors that influence the performance of parallel programs include task start-up time, synchronization, communications, software overhead (compilers, libraries, tools, operating system), and task termination time [67].

More often than not, parallel programming is a long, iterative process, hence the cost of development must also be considered, and there is no point in applying many work hours on a negligible performance enhancement [67]. Parallel programming is, all in all, an art, as there is no "recipe" for parallelization. The developer must use his knowledge about the algorithms and hardware at hand [75].

Communication and I/O

Communication between tasks always incurs in overheads, as resources and processing power that could be used for computation are being used to transmit data and to synchronize (introducing waiting time) between tasks.

One must strive for the so-called **embarrassingly parallel** algorithms, where there is no need for coordination between tasks, or in other words, each task is independent of every other. As an example, many image processing algorithms are embarrassingly parallel, because a large number of operations are per pixel and do not depend on other pixels. Obviously, not all parallel algorithms can be reduced to an embarrassingly parallel version, and communication is needed [76, 67].

Communications between tasks can be synchronous or asynchronous. In **synchronous communications**, also known as **blocking communications**, tasks must wait for the communications to have completed. In **asynchronous communications**, also called **non-blocking communications**, tasks can transfer data independently from one another, or perform work while communications are taking place [67].

File I/O should be reduced, and, if really necessary, the use of a parallel file system is preferable.

Synchronization and Data Dependencies

Synchronization between tasks is needed in two situations: (1) tasks have to join up at a certain point; and (2) to serialize the access to resources in critical sections. Synchronization between tasks can be accomplished by using a few mechanisms [67]:

- **Lock/Semaphore.** Access a critical section in a serial fashion. Only a single task can own the lock/semaphore at a given time and perform operations on the shared resource.
- **Barrier.** Used to synchronize all tasks when reaching a certain point in the code. A task blocks and waits for all others to complete.

There is a **dependence** between program statements when the order of execution affects the results of the program, and this dependence is one of the main culprits to parallelism. A **data dependence** results from the use of the same locations in memory by different tasks. It can be of two kinds: **loop-carried data dependence**, when the computation of a value depends on a value from a previous iteration; or **loop-independent data dependence**, when the same set of variables are read and written by multiple tasks.

In distributed memory architectures, data dependencies are handled by communicating the required data to the other tasks at synchronization points. In shared memory architectures, the problem is solved by synchronizing read/write operations between the tasks.

Occupancy/Load Balancing and Granularity

Work should be distributed among tasks so that all tasks are busy all of the time, avoiding idle times, a metric known as **occupancy**. When a set of tasks encounters a barrier, the performance will be determined by the slowest task. Whenever possible, the work should also be equally distributed among the tasks, achieving **load balancing**. This can be a complicated process, because often, with the same algorithm, the amount of time taken to compute depends on the data [67].

In parallel algorithms, **granularity** measures the ratio of computation to communication. **Coarse-grain parallelism** occurs when large amounts of computational works are done per communication event. This is the desired degree of parallelism for best performance, as the communication overhead is low. In **fine-grain parallelism**, there is a relatively small amount of computational work done between communication events. Due to the high communication overhead, there is less opportunity for performance gains. On the other hand, it is easier to achieve good load balancing with fine-grain parallelism [67].

Chapter 2

Speech Compression

Speech transmission is an important topic in telecommunications, and very important in the context of this dissertation. In this chapter, speech compression will be studied, from the production and perception of human sound, to the raw storage as PCM, and speech compression using many different techniques. The characteristics of the algorithms will be exposed, and DSPs will be analyzed, as they are the usual implementation platform of codecs, besides general-purpose CPUs.

2.1 Properties of Sound

Sound is a sequence of waves of pressure which propagates through a medium (solid, a liquid or a gas) by the movement of atoms or molecules. It is a **longitudinal wave**, that is, the disturbance is in the direction of the wave itself, contrasting with electromagnetic waves and ocean waves, which are transverse waves, which means the oscillations are occurring perpendicular (or right angled) to the direction of energy transfer. As a wave, sound can be characterized by its generic properties [77]:

- **Frequency (or its inverse, the period).** Frequency is perceived as the pitch of a sound. The higher the pitch, the higher the frequency. It is measured in hertz (Hz).
- **Wavelength.** Depends on the wave speed and frequency. Also related to the wavelength is the wavenumber, proportional to the inverse of the wavelength.
- **Amplitude.** Perceived by the ear as loudness, and measured on a logarithmic scale. Amplitude is the air volume displaced by the sound wave.
- **Intensity.** Depends on the energy per unit volume, and on the velocity.
- **Speed.** Depends mostly on the medium it passes through and on temperature. This speed is 343.2 m/s in dry air at 20℃.
- **Direction.**

2.1.1 Production of Speech

There are two important types of sounds produced by air when it passes through the throat, the vocal cords, and the mouth: **voiced sounds**, where the vocal cords vibrate with

a certain frequency of pitch, and is usually made when vowels are pronounced; and **unvoiced sounds**, where the vocal cords don't vibrate, but are opened. This classification of sound types is important because many voice codecs employ this knowledge in the analysis/synthesis process, in order to be able restore the signal according to the original source. The amount of air exhaled during speech determines the amplitude [78, 79].

Speech signals are non-stationary, but the properties of the signal remain relatively constant over short periods of time, between 5 and 20 ms. Voiced sound is quasi-periodic in the time domain, and harmonically structured in the frequency domain, while unvoiced sound has random and broadband characteristics. The statistical and spectral properties that are exploited by the codecs work on these speech segments [78, 79].

Speech signals are generally bandlimited to 4 kHz. According to the Nyquist frequency, sampling must be done at a rate equal to or greater than twice the bandwidth of analog speech, hence a sampling rate of 8 kHz is enough to cover this range [78].

2.1.2 Perception of Sound

Psychoacoustics, the science that studies the sound perception, determines the psychological and physiological responses to sound. The audio compression methods exploit these results by employing lossy compression, eliminating perceptual redundancy by coding with decreased quality, or simply not coding. This redundancy can, for example, be in the form of high frequencies that are very hard to hear or simply unhearable. Much like optical illusions, voice compression algorithms use the brain's ability to form an impression from incomplete information [2].

The human ear can usually hear sounds in the frequency range of 20 Hz to 20,000 Hz, decreasing with age, and up to an amplitude near 100 dB. Most adults are unable to hear above 16 kHz. The human ear is more sensitive to high-frequency sounds.

The brain perceives sounds through 25 to 27 **critical bands**, up to 20 kHz, and the bandwidth for each critical band grows logarithmically with frequency. This operation performed by the brain is called **critical band analysis**. At 100 kHz, the bandwidth is about 160 Hz and at 10 kHz, it is about 2.5 kHz in width [80]. Figure 2.1 shows this concept pictorially. In the case of music or other audio applications, the whole human auditory range must be covered, hence the typical 44.1 kHz sampling rate used in digital music.

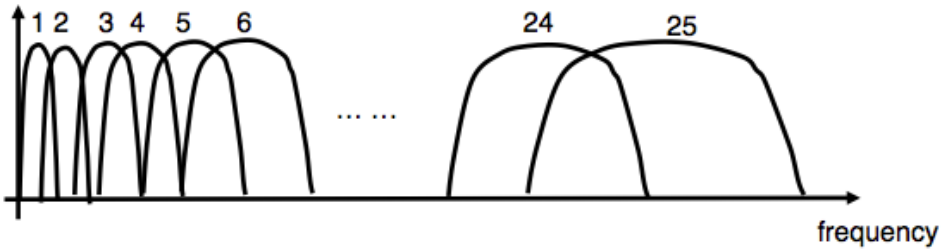


Figure 2.1: Critical bands perceived by the brain (from [80])

The hearing threshold varies with the frequency, and it is most sensitive at around 2 kHz [80]. Figure 2.2 presents the threshold values for the whole hearing range.

Another interesting phenomenon occurring in the human ear is **simultaneous masking**. Masking occurs when a strong signal hides a weaker one, and can occur in the **frequency domain**, or in the **time domain**. The frequency masking allows the hiding of quantification

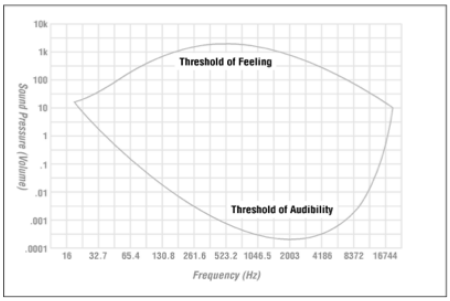
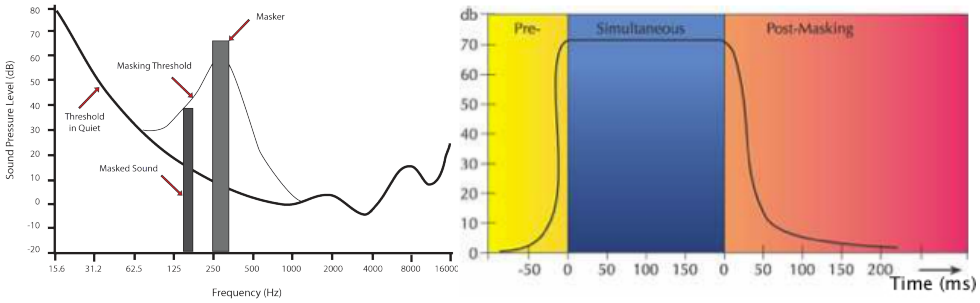


Figure 2.2: Perception threshold (from [80])



(a) Masking effects in the frequency domain (from [81])

(b) Masking effects in the time domain (from [80])

Figure 2.3: Masking effects

noise, aliasing, or transmission errors. The temporal masking of a soft sound can occur before (pre-masking) or after (post-masking) a loud sound. The duration of pre-masking is from 5 to 20 ms, and 50 to 200 ms for post-masking [80]. Figure 2.3 illustrates the masking effects in both domains.

Based on the perception properties, the following can be exploited in audio compression [80]:

- **Subband coding.** Have different quantizations for different critical bands.
- **Encode only audible sounds.** Due to frequency-domain masking, a weaker signal can be discarded if a stronger one exists in the same band. Due to time-domain masking, a soft sound (lower amplitude) can be discarded before or after a loud sound.

  If a stereo system is being used, **stereo redundancy** can be eliminated at low frequencies, since the brain is unable to detect where the sound is coming from, therefore is enough to encode only one channel.

2.2 Coding Techniques

Audio compression is required to meet bandwidth requirements of digital audio streams and to reduce storage size of audio files. This functionality is accomplished by algorithms called **codecs** (a shorthand for COmpression/DECompression, or COder/DECoder). In the speech compression vocabulary, they appear by the name of **speech coders**. Generic data

compression algorithms such as Run Length Encoding, or dictionary-based methods, are not useful for audio compression, reducing the data size by a small percentage, as they do not exploit the audio properties; besides, they are not designed for use in real time applications. Audio compression algorithms, as with image and video compression algorithms, must exploit the nonrandom structure of audio and psychoacoustics, in order to reduce information redundancy [77].

Analog audio, as any other analog phenomena, cannot be processed by digital computers in that form, so a conversion to digital form is needed first. **Pulse-Code Modulation (PCM)** is a method used to represent sampled analog signals in a raw, digital form. PCM has been for many years the de facto standard for digital audio and it is used not only in computers, but also Compact Disc, DVD, Blu-ray and telephone systems.

In order to convert an analog signal originating from a microphone or other device to the digital form, a method known as **modulation**, the signal must be passed through a bandlimiter filter, limited at twice the maximum frequency, by Nyquist Theorem. Then, a **sampling** of the magnitude of the analog signal is made at regular intervals, and each sample is **quantized** to the nearest value within a range of digital steps. The fidelity of the PCM stream to the original analog signal is determined by the **sampling rate/sampling frequency**, the number of times samples are taken per second; and **bit depth**, the number of possible digital values a sample can have. This whole process is accomplished by an **Analog-to-Digital Converter (ADC)**. Figure 2.4 shows the sampling and quantization process for a 4-bit PCM.

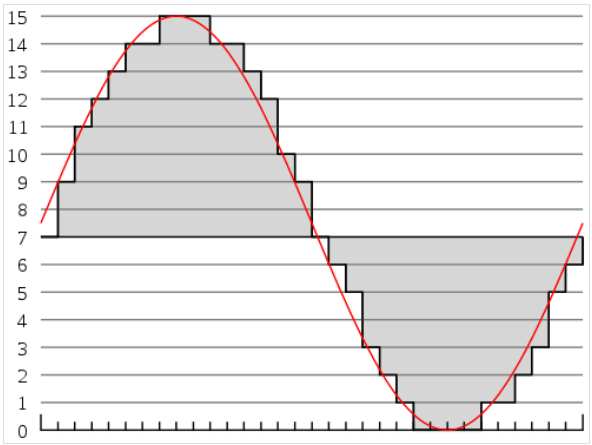


Figure 2.4: Sampling and quantization of a signal (red) for 4-bit PCM (from [82])

PCM data can be characterized by the following attributes [83]:

- **Sampling rate/Sampling frequency.** The number of samples of the sound that get digitized per second. According to the Nyquist-Shannon sampling theorem, the original signal can be perfectly reconstructed if the sampling frequency, F_s, is at least twice the maximum frequency of the signal being sampled. $F_s/2$ is known as the **Nyquist frequency**, and it is the highest frequency that can be represented. Typical values include 8, 16, 32, 44.1 kHz. In telephony, 8 kHz (narrowband) is the norm, but there are codecs that support 16 kHz (wideband), or even 32 kHz (ultra-wideband).
- **Amplitude resolution/bit depth/sample size.** Defines the number of possible

values for the amplitude. For example, if using 8-bit resolution, the amplitude has 256 (2^8) possible values. The most common values are 8 bit ($2^8 = 256$ possible amplitude steps), 16, 20 and 24. In telephony, however, a sample size of 8-bit is the norm.

- **Bit rate.** The number of bits required to encode the speech per unit of time. It can be determined by multiplying the sampling frequency by the sample size, and it is measured in kbit/s, or kbps (not to be confused with kilobytes per second).

- **Sign.** If a sample is 16-bit signed, the sample range is from -32768 to 32767, with a centerpoint of 0. If the sample is unsigned, its range is from 0 to 65535. If the type is signed, the sign encoding usually is in 2's complement. In telephony, the sample is signed.

- **Endianness.** The byte order used to represent the PCM samples (big endian or little endian) must be specified. Since Intel CPUs are little-endian, this is the most common byte orientation.

- **Channels and Interleaving.** Linear PCM encodes a single audio channel, but if more than one channel is sampled, the channels will typically be interleaved: left sample, right sample, left sample, and so on, in a stereo (binaural) system. In telephony, only one channel is used (monaural).

- **Data Type.** Usually PCM samples are stored as integers, but in some cases a floating-point data type is required (samples are in the interval $[-1.0, 1.0]$). In telephony, samples are usually stored as integers, using fixed-point arithmetic on the data.

This discrete, raw PCM signal can be processed by means of a DSP or a general-purpose CPU, where algorithms in many scientific areas, like audio compression, or image processing can be applied to the signal.

The receiving end of the communications circuit converts the binary data back into an analog waveform, a process known as **demodulation**, by a **Digital-to-Analog Converter (DAC)**.

When linear quantization is used, the technique is known as **Linear Pulse-Code Modulation (LPCM)**, or **uniform PCM**. LPCM represents sample amplitudes on a linear scale. With LPCM the values stored are proportional to the amplitudes, contrary to the representation with the logarithm of the amplitude (e.g., companding on A-law and μ-law). As voice signals are mostly low in amplitude, using uniform quantization is extremely inefficient, as it represents the whole range of values with the same quality. Non-uniform quantization is then used, using more bits for lower signal levels. This process is called **companding**, and the signal is not considered to be compressed [84, 78].

There are some variations to the PCM method. In **Differential Pulse-Code Modulation (DPCM)** the encoder stores only the differences between the prediction of the next value, based on previous samples, and the actual value, reducing the number of bits to represent the same information. **Adaptive Differential Pulse-Code Modulation (ADPCM)** complements DPCM with the variation of the size of the quantization step, allowing further bandwidth reduction.

The Logarithmic PCM G.711 A-law and μ-law serve as the basis for all other speech coders. State-of-the art techniques (all of which are lossy) allow for a bitrate of 8 kbit/s with almost no perceptible loss in speech quality [77]. Modern algorithms, such as G.729, have

a good speech quality, while keeping a low bitrate. In exchange, the algorithm complexity increases [85]. The entire speech coding process is schematized in Figure 2.5.

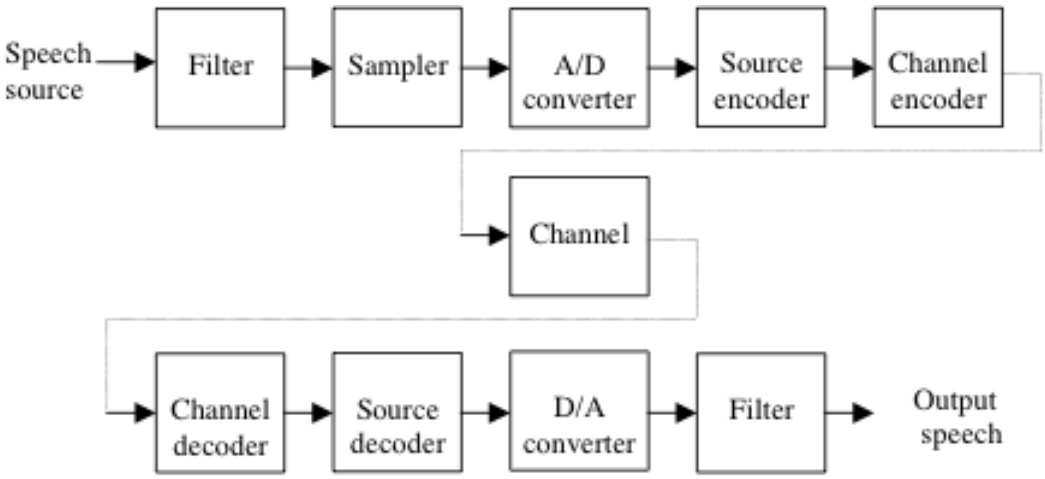


Figure 2.5: Block diagram of a speech coding system (from [79])

Speech coders can be divided into three classes [78, 79]:

- **Waveform coders.** These attempt to preserve the original waveform, producing speech at higher bit rates, and are usually the class of the older codecs. Waveform coders can work in the time domain (PCM, ADPCM) or in the frequency domain (sub-band, adaptive transform).
- **Source coders.** Also known as **vocoders**, or **parametric coders**, source coders assume the speech signal is generated from a model, with some parameters. These parameters are estimated from the input speech signal, and transmitted as the encoded bit-stream. Some source coders operate at very low bit rates and tend to produce speech that sounds artificial and robotic. Linear Predictive Coding (LPC) and Mixed-Excitation Linear Prediction (MELP) are examples of source coders.
- **Hybrid codecs.** Hybrid codecs use techniques from both source and waveform coding, and tend to have intermediate bit rates and good quality. Code Excited Linear Prediction (CELP) coders belong to this class.

Speech codecs always have to make a tradeoff between and are characterized by the following parameters [78, 86]:

- **Bit rate.** Bit rate decrease is the primary objective of speech coding. In general, to achieve high-quality speech coding at low bit rates, high-complexity algorithms have to be used. Low bit rate speech coders typically sound unnatural and synthetic. Toll sound quality usually starts at rates above 16 kbit/s.
- **Complexity.** Codecs can be classified as of low, medium, or high complexity. Codecs such as PCM (G.711 A-law or μ-law) use very little processing power, while G.729 (CS-ACELP), for example, has a high processing requirement. The complexity has an impact on the number of voice channels that can be supported by a processor. Codec complexity is usually measured in Million Instructions Per Second (MIPS).

- **Quality.** The quality of speech given by a codec is determined by a metric called **Mean Opinion Score (MOS)**. MOS are determined by a set of tests performed by listeners (usually 12 to 24 listeners), and are not absolute measures for the comparison of speech coders. They will listen to several speech patterns via different compression techniques, and rate them according to their perceived quality. The highest quality value is 5, and the lowest is 1. Speech quality in digital communications is classified into four categories: broadcast, network or toll, communications, and synthetic.

- **Delay.** Delay is very important for real-time full-duplex communications. The recommendation G.114 [5] places the upper limit of the one-way coding delay at 150 ms, and from 300 ms onwards, the conversation becomes confusing. The **one-way coding delay**, or the elapsed time from the moment a speech sample arrives at the encoder input, to the instant when the same speech sample appears at the decoder output of the receiver, can be decomposed into four major sources: The **encoder buffering delay** occurs when, in order to encode a speech frame, a certain amount of samples have to be collected (typically 160 to 240 frames, or 20 to 30 ms) before the encoding process can take place. The **encoder processing delay** is the time it takes to encode the buffered data, and depends on the processing capabilities of the system. Often, there can be **look-ahead delay**, which is the fixed time it takes to analyze the next frame, in order to help encode the current one. The encoded frame is then subject to **transmission delay**, the amount of time it takes to travel the network and to appear as an input to the decoder. Finally, the decoder introduces **decoder processing delay**, or the time required to produce a frame of synthesized speech [79].

When a coder and decoder use the same algorithm but in the "inverse" order, with the same complexity, it is said to have **symmetrical compression**. Otherwise, if one is more complex than the other, we are in the presence of **asymmetrical compression**.

As mentioned before, the objective of audio compression is to reduce the bandwidth requirements of voice. Capacity planning is essential when building a packet voice network, and for that it is important to be familiar with each codec's specifications, and the bandwidth calculations. Even though the available bandwidth has been increasing, it is still important to compress speech in order to support more clients per channel. As an example, in VoIP, audio is transmitted through a network using a packet format known as RTP, which is transported using UDP, IP and a link layer protocol. All these protocols add considerable overheads during the packetization process.

Some codecs, for bandwidth and energy savings purposes, employ additional techniques such as silence suppression or multi-rate. Silence suppression accomplishes bandwidth savings over 50%, by relying on the **Voice Activity Detection (VAD)** (detects if the voice is active or not) and the **Discontinuous Transmission (DTX)** algorithms (determines, for an inactive voice frame, if it has to send a background noise update). **Multi-rate codecs** encode frames at multiple bit-rates (hence, different quality), depending on the noise and network conditions [87].

Codecs divide speech into blocks, called **frames**, in which, as examined before, the signal has statistical and spectral properties that can be exploited.

Codec specifications organize the encoding/decoding procedures into a series of interdependent stages, or **computational blocks**, with a well-defined purpose, so that it facilitates the implementation on DSPs, making the specification very hard to read and understand [88]. In order to encode a speech frame, these must be run in sequence. Due to space constraints,

the block diagrams for the speech coders will be omitted, and only the commonalities will be exposed. A brief description will be given for each codec, focusing on the computational requirements. Speech coder specifications usually come bundled with a general-purpose ANSI C fixed-point reference implementation.

2.3 Digital Signal Processors

Digital Signal Processors (DSPs) are dedicated integrated circuits usually intended to be present in time-sensitive (often real-time), reliable embedded systems, and typically only need to perform a single application, involving signal processing algorithms. DSPs are cost-effective solutions, with less economic risk than custom hardware, especially important in low volume applications. DSPs have advantage in terms of speed, cost, and energy efficiency, when compared to other types of microprocessors. Their performance and price range is very wide, from low-cost DSPs that are very similar to the early 1980s DSPs to high-speed superscalar DSPs with deep pipelines and extensions such as SIMD [65].

The major DSP manufacturers are Texas Instruments [89], Freescale [90], Analog Devices [91], and NXP Semiconductors [92].

DSPs are usually embedded on a larger system, and perform very specific digital signal processing tasks, while other components handle the general-purpose tasks. Often, DSP applications need to execute in real-time, i.e., their execution latency must be within some limits. It is possible to do digital signal processing on general-purpose microprocessors. However, due to power, space, and cost constraints, general-purpose microprocessors cannot be used in most devices.

DSPs are present in communication devices like mobile phones, modems, fax machines or transcoding cards; multimedia devices such as music players, video cameras, computer sound cards and television sets; battery controllers; car systems such as ABS, electronic fuel injection, and power door locks; aircraft flight control systems; and in scientific and military applications such as radar, sonar; and in seismology. In many of these applications, the DSP acts as a co-processor connected to a host CPU that offloads signal processing work to the DSP. Some of these applications have a high-volume production and require low manufacturing costs.

Typical digital signal processing algorithms require very predictable operation types: additions, multiplications, delays (store a value for later use), and array handling. Therefore, DSP architectures exploit this predictability. The real-time requirements of a system mean that there is a hard restriction on the response time to one or more events, e.g.: the arrival of new input samples or the requirement that a new output sample be generated. In DSPs, important operations are often pre-calculated and stored in ROM as static lookup tables, and algorithms only have to get the result from this lookup table, allowing for faster, more predictable computations. Sources of indeterminacy, like branch predication, data-dependent termination of functions, and memory management procedures may put the realtime constraint at risk, so the program must always allow for worst case [93, 94].

Speech coders are typically implemented in DSP platforms. DSPs meet the speech processing requirements, as they have an architecture optimized for signal processing applications, especially MAC units with support for saturation, at an attractive area, cost, and power per channel ratio, and with good development toolchains [95, 96]. As seen in Chapter 1, audio transcoding cards for Asterisk generally employ DSP chips.

One early decision in the speech coding industry was to whether the algorithms would run on fixed-point or floating-point platforms. Fixed-point numbers (not to be confused with integers) have a limited dynamic range when compared to floating-point numbers. On the other hand, one can represent very small numbers and very big numbers using floating-point numbers. Due to the minimization of rounding errors, hence reduced risk of numeric overflows, and increased speed, floating-point implementations should be regarded as the optimal choice. However, floating-point processors need more complex and expensive hardware, as well as larger power and space requirements, and fixed-point implementations were adopted as the norm [79, 65].

DSPs support a limited amount of memory with special addressing modes such as circular addressing (for filters), or bit-reverse addressing (for FFT). Like regular computer microprocessors and microcontrollers, DSPs have on-chip peripherals such as serial port controllers and timers. DSPs are generally programmed once, using C or Assembly, and the program is stored in ROM, the so-called **firmware**.

DSPs generally don't use the conventional von Neumann architecture typically used in general-purpose microprocessors. Instead they use memory architectures that allow the fetching of multiple data and/or instructions at the same time, compromising flexibility as the memories are not interchangeable, but increasing performance.

Although digital signal processing generally works with analog signals, they have to be digitized to work with them. DSPs require the signal to be discrete. The device that converts analog signals to digital form is the ADC. The discrete signals are then able to be processed by the DSP. After that, depending on the application, they can be converted again to the analog form, using DACs.

As DSP chips aren't usually present by themselves, I/O technologies are integrated in them. There are many serial or parallel interconnect technologies supported by DSP processors: SRIO (for interconnecting embedded processors), HYPER LINK, PCIe, LAN, and USB, along with DMA controllers that reduce the memory transfer overheads [65].

General-purpose CPUs architectures, like x86 and PowerPC, have hardware acceleration for signal processing tasks. The most common extensions for signal processing are the SIMD-based instruction sets, such as MMX and SSE on x86, and AltiVec on the PowerPC [65].

2.3.1 Features and Algorithms

Digital signal processing has a set of well established techniques. Due to the predictability of the operations, they can be hardware-accelerated. DSP architectures are always evolving, meeting the ever-changing needs of DSP algorithms. Nearly every feature on a DSP is backgrounded by algorithms whose computation is accelerated by the inclusion of this feature. DSPs are usually regarded as the best platform for digital signal processing implementations, and are also applied in modulation and demodulation techniques, allowing for less power consumption and lower costs [97, 98].

DSP instruction sets are made so that the underlying hardware features are easily expressible to the programmer in Assembly, since they aren't frequently available in the higher-level C language. In addition, the instructions generally allow the specification of several parallel operations in a single instruction (VLIW), like one or two data fetches from memory (with address pointer updates) in parallel with the main arithmetic operation. This allows as well the programs to remain smaller, as the applications are cost-sensitive and memory is limited [65, 93].

The most commonly used operation in DSP algorithms is the **Multiply-Accumulate (MAC)** in the form:

$$accumulator \leftarrow accumulator + X * Y$$

DSPs are optimized for MAC performance, as many DSP algorithms use these types of arithmetic operations. Many DSPs have several MAC units in a parallel fashion. If the operation is floating-point based, it is called **Fused Multiply-Add (FMA)** or **Fused Multiply-Accumulate (FMAC)**. This operation is used, for example, in FIR filters (used to remove or enhance parts of a signal), or matrix multiplications (used in video compression). Several optimizations have been made since the inception of dedicated MAC units, as well as the popularization of the modified Harvard architecture, allowing single-cycle execution of MAC operations [93]. As will be seen in Section 2.4, it is also a very important operation in speech coding.

Contrary to regular x86 microprocessors' modular arithmetic, DSPs have what is called saturation arithmetic. **Saturation arithmetic** prevents an operation result from overflowing, by setting a maximum and a minimum range, to which any result that is greater than the maximum, or is below the minimum, is clamped to, avoiding overflows. In other words, it performs **clipping** on a waveform. This is extremely useful for audio compression, as one of the most used operations is ensuring the signal is within some range, avoiding catastrophic loss in signal-to-noise ratio. In x86 architectures, saturation registers are only available on the MMX or SSE extensions. General-purpose registers only work in wrap-around mode, that is, when a result generates an overflow, it wraps around to the minimum value, which is very damaging to the signal-to-noise ratio in a DSP system. It is possible to implement integer saturation arithmetic in general-purpose CPUs. However it requires branching, since the signal will be tested against the maximum and minimum values [99].

DSPs often have a large amount of registers, as many algorithms require that intermediate values be held for later use [100].

Signal processing algorithms spend the vast majority of their processing time in loops. DSPs exploit this characteristic by providing **zero-overhead looping**, special loop instructions that ensure that no cycles are expended updating and testing the loop counter, or branching back to the top of the loop [65].

2.4 Speech Coders

3GPP and 3GPP2 specify that all 3GPP IMS terminals must support the AMR speech codec, while terminals providing wideband services must support the AMR-WB codec [1]. Due to the lack of proficiency in the signal processing techniques required by the coders, only a computational perspective will be taken. In order to expose the computational commonalities between speech coders, codecs for other recommendations, such as H.323 and IMT-2000, will be analyzed, through higher-level descriptions and analysis of existing implementations.

2.4.1 ITU-T G.711

G.711 [84], formally "Pulse code modulation (PCM) of voice frequencies", is an ITU-T standard for fixed-point audio companding (a portmanteau between compressing and expanding) introduced by Bell Systems in 1972 and standardized in 1988. G.711 is required in H.320

[101] and H.323 [6] recommendations and is used in PSTN and VoIP [2]. Talking about PCM in the context of telephony, is talking about G.711. G.711 serves as a base for all other codecs.

G.711 works at a sampling rate of 8 kHz, which, by Nyquist theorem, means it can encode frequencies from 0 up to 4 kHz. Its frame size is 0.125 ms, and the typical algorithmic delay is 0.125 ms, as there is no look-ahead delay. This codec has the lowest latency because there is no compression.

DSP implementations using C of G.711 can be seen in work by Embree [94] and is discussed by Texas Instruments in [102]. General-purpose implementations are available, such as FFmpeg [103], Ekiga [104], Asterisk [30], and ITU-T Recommendation G.191 [105].

G.711 has two variants: μ-law, used in the United States and Japan; and A-law, used in the rest of the world. μ-law is related to the T1 standard, while A-law is related to the E1 standard. The difference between the two methods lies on the sampling method for the analog signal. Both use a technique known as **Logarithmic PCM**, by doing a non-uniform quantization for each sample. Lower signal values are encoded using more bits, while higher signals require fewer bits. The consequence is that low amplitude signals are well represented, while having enough dynamic range to encode high amplitudes. The sound quality of A-law is better as it provides more dynamic range [106]. With logarithmic PCM, a 14- (A-law) or 13-bit (μ-law) linear PCM sample value is mapped into an 8-bit value, creating a high bit rate 64 kbit/s stream.

G.711 requires very little CPU power (near zero). The transcoding operations between G.711 and linear PCM can be reduced to simple table lookups. The encoding does not use logarithmic functions, but approximations. The input range is divided into segments, and each segment uses a different interval between decision values. Most segments contain 16 intervals, and the interval size doubles from a segment to the next one [107, 2].

The encodings are symmetrical around zero. In μ-law, there are 8 segments of 16 intervals in both the positive and negative directions, while in A-law there are 7 segments.

The decoding procedure follows the same approach, by mapping 8-bit values to 13- or 14-bit uniform PCM samples.

2.4.2 ITU-T G.722

G.722 [108], "7 kHz audio-coding within 64 kbit/s", is an ITU-T standardized fixed-point audio codec for high quality telephony. It was approved in November 1988, and was the first wideband speech codec standard, with the intent of being used in teleconferencing, commentary channels, and ISDN telephony applications [109].

Using the **Sub-Band Adaptive Differential Pulse-Code Modulation (SB-ADPCM)** technique, it separates the signal into two 4 kHz frequency bands and encodes each one of the bands independently using ADPCM. The higher band is always quantized with two bits, while the lower band is quantized with six, five, or four bits, depending on the selected mode.

G.722 is designed for the 50 Hz to 7000 Hz audio range, and works at a 16 kHz sampling rate. The G.722 coder operates on 10 ms speech frames, each with 160 samples. Its frame size (delay) is 10 ms, look-ahead delay is 0.125 ms, resulting in a total algorithmic delay of 10.125 ms.

G.722 has three selectable bit rates: 48, 56, or 64 kbit/s, but in practice it always uses 64 kbit/s. By using the latter two modes, 8 and 16 kbit/s auxiliary data can be encoded within the 64 kbit/s, by making use of bits from the lower sub-band.

DSP implementation discussions can be found in work by Peretz and Gumusoglu [110] (master's thesis), Lai et al. [111], Embree [94], and Makelainen [109]. General-purpose implementations include ITU-T Recommendation G.191 [105], and SpanDSP [112] (used in Asterisk).

During the encoding procedure, the 16 kHz signal passes through the transmit Quadrature Mirror Filters (QMFs), two linear-phase non-recursive digital filters which split the initial frequency band, 0 to 8000 Hz, into two sub-bands: the lower sub-band (0 to 4000 Hz) and the higher sub-band (4000 to 8000 Hz), sampled at 8 kHz. The lower sub-band signal is encoded by the lower sub-band ADPCM encoder. On this encoder, the lower sub-band signal is subtracted an estimate of the input signal, producing the difference signal. By using an adaptive 60-level non-linear quantizer, six bits are assigned to the value of the difference signal, producing a 48 kbit/s signal. The two least significant bits of the quantized signal are deleted, creating a 4-bit signal which is used for the quantizer adaptation and applied to a 15-level inverse adaptive quantizer to produce a quantized difference signal. The signal estimate is added to this quantized difference signal to produce a reconstructed version of the lower sub-band input signal. The feedback loop is completed by an adaptive predictor which operates upon the reconstructed signal and the quantized difference signal, producing an estimate of the input signal. The higher sub-band signal is encoded by the higher sub-band ADPCM encoder. On this encoder, the higher sub-band signal is subtracted an estimate of the input signal, producing the difference signal. By using an adaptive 4-level non-linear quantizer, two bits are assigned to the value of the difference signal, producing a 16 kbit/s signal. A quantized difference signal is produced by an inverse adaptive quantizer, from the same two bits. The signal estimate is added to the quantized difference signal to produce a reconstructed version of the higher sub-band input signal. The feedback loop is completed by an adaptive predictor that operates upon both the reconstructed signal and the quantized difference signal, producing an estimate of the input signal. The outputs of the lower and higher sub-band ADPCM encoders are combined using the multiplexer, producing a 64 kbit/s signal for transmission [108].

During the decoding procedure, the received 64 kbit/s signal is first decomposed into two signals, forming the codeword inputs to the lower and higher sub-band ADPCM decoders, respectively. The lower sub-band signal is decoded by the lower sub-band ADPCM decoder. On this decoder, an estimate of the input signal, including the quantizer adaptation, is generated by the same feedback portion of the lower sub-band encoder. The reconstructed signal is produced by adding to the signal estimate a quantized difference signal (depending on the mode). The higher sub-band signal is decoded by the higher sub-band ADPCM decoder. The reconstructed signal is generated by the same feedback portion of the higher sub-band ADPCM encoder. Finally, through the receive QMFs, the 16 kHz output signal is formed by two linear-phase non-recursive digital filters which interpolate the outputs from the lower and higher sub-band ADPCM decoders from 8 kHz to 16 kHz [108].

2.4.3 ITU-T G.726

G.726 [113], formally "40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)" is a multi-bitrate voice codec standardized by International Telecommunication Union - Telecommunication Standardization Sector (ITU-T).

G.726 uses the ADPCM technique. The result is a quality almost as good as G.711, but using only half of the bandwidth [2].

Its frame size (delay) is 0.125 ms, and it does not wait for the next sample, so it has a look-ahead delay of 0 ms. G.726 can work at 16, 24, and 32 kbps, but Asterisk only supports 32 kbps, as it is the most popular rate [2].

Implementations using fixed-point DSPs are discussed by Wang et al. [114], Sharma et al. [115], and Vehvilainen et al. [116]. General-purpose processor implementations are featured in ITU-T Recommendation G.191 [105], and Asterisk.

The encoder takes as input an A-law or μ-law PCM signal, and the signal is converted to uniform PCM. After the conversion, a difference signal is calculated, by subtracting an estimate of the input signal from the input signal itself. The value of the difference signal is assigned by an adaptive 31-, 15-, 7-, or 4-level quantizer with five, four, three, or two bits, respectively, depending on the encoder bit rate. The signal estimate is added to the quantized difference signal to produce the reconstructed version of the input signal. An adaptive predictor operates upon the reconstructed signal and the quantized difference signal, producing an estimate of the input signal, completing the feedback loop [113].

The decoder is structurally identical to the encoder's feedback portion, with a uniform PCM to A-law or μ-law converter and a synchronous coding adjustment.

2.4.4 ITU-T G.729

G.729, formally "Coding of speech at 8 kbit/s using CS-ACELP" is an ITU-T standardized fixed-point modern speech compression algorithm for high quality and low bandwidth telephony [117]. As it is a relatively recent codec, its patents are still in force. Nonetheless, it has become one of the most popular speech codecs [118].

CELP derivatives, as low bit rate techniques, are useful in environments with a great amount packet loss and delay, such as mobile applications [119]. G.729 is used in many applications, for example VoIP, VoN, VToA, ISDN [85, 120]. Its major selling point is the quality it can deliver, while keeping the bandwidth down. However, the processing power required is high. The quality is equivalent to that of 32 kbit/s ADPCM [85].

G.729 codes speech for multimedia applications at a bit rate of 8 kbit/s, using a technique known as **CS-ACELP**, a technique based on CELP. G.729 operates on 10 ms speech frames, each with 80 samples, from a 8000 Hz, 16-bit linear PCM source.

DSP implementations are discussed by Kong et al. [118], Kun et al. [121], Swee [86], Sheikh et al. [85], Arora et al. [122], Kim et al. [123], and Najafzadeh et al. [124]. General-purpose implementations are discussed by Wang et al. [125] (ARM).

During the encoder procedure, the input signal first goes through a high-pass filter and is scaled in the preprocessing block. Then, LP analysis is done once per 10 ms frame to compute the LP filter coefficients. The coefficients are converted to Line Spectrum Pairs (LSPs), and quantized using predictive two-stage Vector Quantization (VQ) with 18 bits. Via an analysis-by-synthesis search procedure, the excitation signal is chosen, in which the error between the original and the reconstructed speech is minimized according to a perceptually weighted distortion measure. This is accomplished by filtering the error signal with a perceptual weighting filter, with coefficients derived from the unquantized LP filter. The excitation parameters (fixed and adaptive-codebook parameters) are determined for 5 ms (40 samples) subframes. The quantized and unquantized LP filter coefficients are used for the second subframe, and in the first subframe, the quantized and unquantized interpolated LP filter coefficients are used. Based on the perceptually weighted speech signal, an open-loop pitch delay is estimated. The following operations are done for each subframe. The target signal is computed by filtering

the LP residual through a weighted synthesis filter, involving a weighting filter, and a LP synthesis filter. These filters' initial states are updated by filtering the error between LP residual and excitation. Afterwards, the impulse response of the weighted synthesis filter is computed. In order to find the adaptive-codebook delay and gain, a closed-loop pitch analysis is done using the target signal and the impulse response. The target signal is updated by subtracting the adaptive-codebook contribution, and the new target signal is used in the fixed-codebook search to find the optimum excitation. An algebraic codebook with 17 bits is used for the fixed-codebook excitation. The adaptive and fixed-codebook contributions gains are vector quantized with 7 bits. Ultimately, the filter memories are updated using the determined excitation signal [117].

On the decoder, the parameter's indices are first extracted from the received bit stream. The indices are decoded in order to obtain the coder parameters corresponding to a 10 ms speech frame. The parameters are the LSP coefficients, the two fractional pitch delays, the two fixed-codebook vectors, and the two sets of adaptive and fixed-codebook gains. The LSP coefficients are interpolated and converted to LP filter coefficients, for each subframe. The following steps are done for each 5 ms subframe. By adding the adaptive and fixed-codebook vectors, scaled by their respective gains, the excitation is constructed. By filtering the excitation through the LP synthesis filter, the speech is reconstructed. The reconstructed speech signal is run through a post-processing stage, including an adaptive postfilter based on the long-term and short-term synthesis filters, followed by a high-pass filter and a scaling operation [117].

2.4.5 GSM-FR

GSM Full Rate (GSM-FR) [126], also known as GSM 06.10, or simply GSM, is one of the four codecs in the GSM standard, the European digital mobile cellular telephone system. The others are GSM-EFR, GSM-HR, and AMR.

GSM-FR codes speech at a 13 kbit/s bit rate, using a technique called Regular Pulse Excitation-Long Term Prediction (RPE-LTP). GSM-FR operates on 20 ms (160 samples) frames, sampled at 8000 Hz.

DSP implementations are discussed by Nurmi et al. [127] and Owall et al. [128]. Asterisk has a general-purpose implementation of GSM-FR.

First, the signal goes through a preprocessing stage, to eliminate the signal's DC component, implemented as a first-order FIR filter. On the LPC Analysis stage, the speech signal is divided into 20 ms (160 samples) segments, and an LPC analysis is run on each segment, calculating eight reflection coefficients using the Schur recursion algorithm. These reflection coefficients are then converted into Log Area Ratios (LARs), by a piecewise linear approximation. The speech frame is divided into 4 sub-frames with 40 samples of the short term residual signal in each, and the sub-frames are processed blockwise by the subsequent elements. The parameters of the long term analysis filter, the LTP lag and the LTP gain are estimated and updated in the LTP analysis block, before the processing of each sub-block of 40 short term residual samples. A block of 40 long term residual samples is obtained by subtracting 40 estimates of the short term residual signal from the short term residual signal itself, and it is fed to the RPE analysis. The RPE analysis determines a representation of the 40 input long term residual samples through one of 4 candidate sub-sequences of 13 pulses each. The pulses are encoded using ADPCM, and the parameters are fed to an RPE decoding and reconstruction module, producing a block of 40 samples of the quantized version of the

long term residual signal. A reconstructed version of the current short term residual signal is obtained by adding these 40 quantized samples to the previous block of short term residual signal estimates. Finally, the feedback loop is completed by feeding the block of reconstructed short term residual signal samples to the long term analysis filter, producing a new block of 40 short term residual signal estimates [126].

The decoder follows the same feedback loop structure of the encoder, outputting the reconstructed short term residual samples. The final reconstructed speech signal samples are achieved by applying a short term synthesis filter followed by a de-emphasis filter to the reconstructed short term residual samples [126].

2.4.6 GSM-HR

GSM Half Rate (GSM-HR) [129] is standardized by ETSI for GSM, and based on the Vector Sum Excited Linear Prediction (VSELP) technique, attaining a speech quality equivalent to GSM-FR, at a lower 5.6 kbit/s bit rate, but with increased complexity.

The encoding process is done on 20 ms speech frames. GSM-HR takes as input a 13-bit PCM signal from the audio part, from the network, or from PSTN by converting 8-bit A-law or μ-law to 13-bit uniform PCM. GSM-HR processes 20 ms frames, with 160 samples each, at a time, and these frames are further divided into four 5 ms (40 samples) sub-frames. The end result is a 5.6 kbit/s bitstream.

A DSP implementation discussion can be seen in the work by Lahane et al. [130]. A general-purpose C reference implementation is provided by ETSI.

The input speech is initially filtered by a 4^{th} order pole-zero high-pass filter (two second-order IIR filters), in order to suppress the frequencies below 120 Hz. Then, once per frame, the LPC filter coefficients are computed, by applying a fixed-point covariance lattice technique. Using analysis-by-synthesis, the code to use to represent the excitation for each subframe is determined. There is a codebook search procedure, in which the coder tries each codevector as a possible excitation for the CELP synthesizer. The synthesized speech is then compared with the input speech, generating a difference signal. The difference signal is filtered by spectral weighting filters, and the power is computed on a weighted error signal. Based on the minimum weighted error power, the codevector is chosen for that subframe, and the frame energy is computed and coded once per frame. Once per frame, a voicing mode is selected, and in case of a voiced frame, a long-term predictor is used, and a VSELP codeword is selected [129, 130].

The decoder creates the combined excitation signal from the long term-filter state and VSELP codevector. For the unvoiced mode, the long-term filter state is replaced by another VSELP codebook and the pitch pre-filter is not applied. The combination excitation is processed by an adaptive pitch and gain. Then, the prefiltered excitation is applied to the LPC synthesis filter. Finally, after reconstructing the speech signal using this synthesis filter, an adaptive spectral postfilter is applied, as well as an automatic gain control [129, 130].

2.4.7 MELP

Mixed-Excitation Linear Prediction (MELP) [131] is the United States Department of Defense (DoD) adopted coder for 2400 bps voice communication. The MELP coder is based on the traditional LPC technique, with a few improvements: mixed excitation, aperiodic pulses, adaptive spectral enhancement, pulse dispersion filtering, and Fourier magnitude

modeling.

The MELP coder has a 2.4 kbit/s bit rate. The frame size is 22.5 ms, corresponding to 180 voice samples at a 8 kHz sampling rate.

A DSP implementation discussion can be found in work by Lin et al. [132].

During encoding, the input samples are first filtered by a 4^{th} order Chebyshev Type II high-pass filter, to remove the DC component. Then, the samples are further filtered by a 1 kHz 6^{th} order Butterworth low-pass filter, in order to estimate a rough pitch lag, and the final pitch is obtained by a fractional pitch refinement procedure. After determining the final pitch, the signal power or gain is then estimated. Two input gain values are measured (in dB), using a pitch adaptive window length, forming the Root Mean Square (RMS) values of the input signal over the time window. After this process, the signal is filtered into five frequency bands: 0-500, 500-1000, 1000-2000, 2000-3000, and 3000-4000 Hz. Pitch analysis is performed on each band to determine if that band is voiced or unvoiced. On the next analysis stage, a 10^{th} order Linear Prediction (LP) analysis is performed on the input speech, using a 200-point (corresponding to 25 ms) Hamming window, centered on the last sample of the current frame. By using a prediction filter, the LPC residual signal is calculated, using the coefficients determined by the LP analysis. In the case of voiced speech, there is a grouping of the magnitudes of the first ten pitch harmonics, using a 512-point FFT on 200 samples run on the LP residual. The parameters to be quantized are: Line Spectral Frequencies (LSF), pitch lag, gain, bandpass voicing, and Fourier magnitudes. If the frame is unvoiced, forward error correction codes are added [132].

When decoding, the decoder unpacks the received bits and convenes them into the parameter code words. The decoding procedure depends on whether the signal is voiced or unvoiced. This is determined by decoding the pitch. If the pitch code is equal to zero or only has one bit set, then the mode is unvoiced, and the decoder performs error correction, and the parameters assume default values. If 2 bits are set in the pitch code, then a frame erasure is indicated. If none of these conditions apply, the pitch value and other parameters are decoded, as it is the voiced mode. The MELP synthesis parameters are then interpolated pitch-synchronously, for each synthesized pitch period, including the LSFs, log gain, pitch, jitter, Fourier magnitudes, pulse and noise coefficients for mixed excitation, and spectral tilt coefficients for the adaptive spectral enhancement filter. Through the sum of the filtered pulse and noise excitations, the mixed excitation is generated. By applying an IDFT of one pitch period in length, the pulse excitation is attained. Using a uniform random number generator, the noise signal is generated. The mixed excitation is formed by filtering and summing the pulse and noise excitation signals, which is then filtered by an adaptive spectral enhancement filter. This filter is a 10^{th} order pole-zero filter, accompanied by an additional 1^{st} order spectral tilt compensation. The coefficients are determined by bandwidth expansion of the LP filter transfer function. The LP synthesis stage uses a direct form filter, with the coefficients corresponding to the interpolated LSFs. The gain value is applied to the synthesized speech. The scaling factor is computed for each pitch period, and linearly interpolated to prevent discontinuities in the synthesized speech. Finally, pulse dispersion is applied, formed by a 65^{th} order FIR filter, derived from a spectrally flattened triangle pulse [132].

2.5 Characteristics of Speech Coders

Speech compression algorithms, similarly to most of the other digital signal processing algorithms, are MAC and looping intensive. DSPs reflect the characteristics of these algorithms, as they are optimized to achieve best performance on them, and use their capabilities: single cycle MAC or sometimes multiple MAC units, zero overhead looping, specialized addressing modes and hardware support for fixed point arithmetic, all using low power consumption and reduced costs [133, 134].

Through the inspection of the specifications and implementations, speech coders can ultimately be reduced to the following building blocks:

- **Simple arithmetic.** Shifts, additions and subtractions. Additions and subtractions frequently require underflow and overflow control with saturation. Saturation using flow control instructions is needed on architectures without saturation hardware.
- **Quantization.** Maps a large set of input values to a smaller set, via lookup tables.
- **Filters.** Digital filters can be recursive (IIR) or non-recursive (FIR) [135].
- **Matrix Operations.** Levinson-Durbin recursion.
- **Transforms.** DFT and IDFT.
- **Codebooks.** A codebook is a list of vectors (or codevectors), used for matching patterns and replacing them with shorter representations. Fixed and adaptive codebooks require static or dynamic storage. The codebook is accessed by a search algorithm that can be computationally complex [77].

There hasn't been much research and released standards on the speech coding subject in recent years. No alternative implementations, e.g.: replacing some lookup tables with pure computation, were found for any of the coders.

Coders are very complex, and often require many man-months or even man-years just to adapt to a new platform and to test them, even if the supplied C code is used [96, 130]. Besides, the fixed-point implementations in general-purpose CPUs are usually very confusing and diverge from the codec specifications.

2.6 Optimizations

The algorithms described in the specifications are often too complex (in terms of MIPS) for real-time implementation on certain DSPs, and profiling is used to determine the critical parts of the algorithms [86, 136]. The techniques shown in this section, found throughout the literature, are employed to optimize or reduce complexity of speech coders, and are valid for all speech codecs on any DSP platform [133, 119, 134, 124, 137, 138].

Optimizations regarding data types and variables include:

- **Integers.** Use unsigned integers instead of signed integers when possible.
- **Division and remainder.** Rewrite them as multiplications, which are much faster.

- **Combining division and remainder.** When both division and remainder operations are performed on the same data, the compiler can combine them and perform only a division.
- **Global variables.** Avoid global variables, as they are never allocated to registers.
- **Register optimization.** Use registers for temporary variables (using the `register` C keyword), avoiding slower reads/writes from main memory.
- **Saturation and rounding arithmetic hardware.** Use these hardware mechanisms in order to avoid overflows, without branching.
- **Floating-point arithmetic.** For some algorithms, the use of floating-point operations is required. Floating-point divisions are typically twice as slow as additions or multiplications, and should be rewritten as multiplications. Float variables require less memory and fewer registers than doubles, and their reduced precision makes them more efficient. Transcendental functions like trigonometric functions, logarithm and exponential functions should be avoided as they are usually ten times as slow as a multiplication.

Loops are widely used in speech coders, and a significant amount of execution time is spend in them. The following optimizations can be made:

- **Loop unrolling.** Repeating the body of the loop with different indices. This increases ALU usage, and reduces the loop costs by avoiding branching and variable assignments, but increases the code size. Useful for small loops.
- **Loop merging.** Combining several loops into a single one, reducing the number of operations.
- **Loop splitting.** Splitting a large loop with several variables into shorter loops, improving register usage.
- **Hardware loops.** Use the hardware loop mechanisms present in many DSPs, incurring in zero overhead branching.
- **Loop termination.** The termination condition on loops can cause significant overheads. Thus, simple termination conditions and countdown-to-zero loops should be used.

Operations involving pointers can be optimized as follows:

- **Pointer addressing.** Using pointers instead of indices to access array elements, avoiding the address calculation overheads.
- **Structures.** Pass them by reference, to avoid the copy of the entire structure to the stack.

Functions optimizations include:

- **Intrinsic functions.** Instead of the regular arithmetic functions, use optimized versions for the architecture being used. These are not available in regular ANSI C/C++ libraries.

- **Minimize function call overheads.** Function call overheads can be minimized by reducing the number of parameters passed.
- **Inline functions.** Replace frequently used function calls by the body of the function. This avoids function call and return overheads, at the expense of a larger code size.
- **Leaf functions.** Prioritize the use of leaf functions, functions that do not call any other functions. These are compiled very efficiently on all platforms.
- **Assembly.** Rewrite functions in Assembly if necessary, manually optimizing resource usage.
- **Algorithmic reduction.** If it is possible, reduce the complexity of the algorithm.
- **Pipelining.** Instruction dependency should be minimized, allowing an efficient usage of the processor's pipelining mechanism.

The only optimizations found for general-purpose processor implementations focus on reducing the loop overheads and reducing memory accesses [133].

Chapter 3

GPGPU

This chapter introduces the GPGPU parallel programming implementation. A historical review will be made, showing how the computer graphics industry turned into general-purpose computing. The two most popular GPGPU implementations, CUDA, and OpenCL will be analyzed, but focusing mainly on CUDA. Then, a review on digital signal processing using GPUs will be made.

3.1 History

The history of using the GPU as a co-processor began in the mid-1980s. In 1985, IBM released one of the first 2D graphics accelerators for the IBM PC, the IBM Professional Graphics Controller, which was years ahead of the competition, but it was a commercial failure.

The following year, Commodore released the Amiga 1000, an affordable computer which featured a custom chip for graphics acceleration, offloading most video functions to hardware, including line drawing, area fill and BLIT operations. It was a commercial success, especially in the video editing market.

In the early 1990s, S3 Graphics introduced the first single-chip 2D accelerator.

By the mid-1990s, all major personal computer graphics chip makers had integrated 2D acceleration to their chips. Video game consoles such as Sony PlayStation, Sega Saturn and Nintendo 64 started popularizing 3D graphics in games. In the film industry, computer-generated effects and characters became mainstream, first with Terminator 2, and the first feature film made entirely with computers, Toy Story.

On the personal computer market, dedicated 3D accelerator cards appeared, offloading 3D operations from the CPU. One such example is the Voodoo series by 3dfx. Companies like NVIDIA, ATI, Matrox and Rendition embraced single-chip integrated 2D/3D solutions.

The Voodoo2 chip brought Scan-Line Interleave (SLI) to the market. With this solution, users could have two graphics boards connected, each drawing half the scan lines of the screen. This technology was, with the acquisition of 3dfx by NVIDIA, absorbed and used in newer products by NVIDIA. ATI now also has a multi-GPU technology named CrossFire.

In 1999, NVIDIA launched the GeForce 256, while coining the term Graphics Processing Unit (GPU) [139]. This chip introduced hardware transform and lighting (T&L). With this capability, transforms (moving 3D objects in a virtual world, and the conversion to a 2D space), clipping (drawing visible objects only) and lighting (calculation of color based on

light equations) could be performed on the graphics board, which proved to be an advantage when using slow CPUs.

In 2001, NVIDIA released the first graphics chip with programmable pixel and vertex shaders [140]. This was a very important introduction, as GPGPU is built on top of these programable units.

In 2002, NVIDIA introduced "Cg", C For Graphics [141], allowing higher level pixel and vertex shader programming, where previously developers had to use Assembly language. High-level shading languages later became standardized with Microsoft DirectX's HLSL and OpenGL's GLSL. Researchers and developers started using these programmable features for non-graphics tasks, although this was a very complicated process. This marked the dawn of GPGPU.

By the time programmable pipelines were introduced, with their vertex and fragment shader processors, and shader language compilers (assembly code to hardware instructions) began to be released, researchers started harnessing the GPU for non-graphics tasks. This approach had many limitations and the task of programming for GPUs was an onerous one, as the programmers had to "disguise" their applications as graphics applications, using Direct3D and OpenGL APIs, along with shading languages. It was a very tedious process, and most programmers with the interest in parallel computing were not experts in graphics programming. The APIs allowed to work in terms of shaders, textures and fragments, while programmers in the parallel world wanted streams, kernels, scatters, and gathers. However, the proof of concept was successful, and encouraged the development of modern GPGPU [142].

In 2006, the Sony PlayStation 3 video game console was released, employing a novel microprocessor architecture known as Cell Broadband Engine [143]. This architecture was co-developed by Sony, Sony Computer Entertainment, Toshiba, and IBM, through an alliance named STI, formed in 2001. The basic configuration combines a PowerPC-based core (called PPE) with eight coprocessors (SPEs), and a circular data bus to connect the different elements (EIB). Each SPE is a RISC processor with a 128-bit SIMD organization, allowing it to process sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, or four single-precision floating-point number in a single clock cycle. With these coprocessors, the floating-point performance is much larger than that of general-purpose microprocessors.

The first high-level languages and libraries for GPGPU were built on top of the DirectX and OpenGL graphics APIs. Sh [144] was an early GPGPU implementation. Stanford's Brook [145, 146] was another early implementation of General-Purpose computation on Graphics Processing Units (GPGPU) created as an extension to the C language, "C with streams". It presented the GPU as a easier-to-program coprocessor, abstracting from the DirectX, OpenGL, and x86 implementations. This was the first high-level GPGPU API to earn public praise. NVIDIA and ATI readily hired some of the researchers involved, and both introduced unified architectures.

RapidMind (acquired by Intel in 2009) was a commercial successor to Sh [147]. PeakStream (acquired by Google in 2007) released a parallel processing library for ATI GPUs in 2006 [148].

ATI released Close To Metal (CTM) [149] in 2006, dropping DirectX and OpenGL APIs for a lower-level approach. ATI Stream, CTM's commercial successor was released in December 2007. Since then, AMD has discontinued these products and now supports only OpenCL.

In 2006, NVIDIA introduced the GeForce 8800 series (codenamed G80) architecture [150], featuring a unified shader architecture. Pixel and vertex shaders ceased to be separated

functional units, and a set of general-purpose floating point processors (stream processors) was introduced. With it, NVIDIA introduced the CUDA architecture for the GeForce 8800, claiming performance gains up to 100 times the CPU approach for certain algorithms, and an ANSI C language compiler for the GPU, CUDA C [151].

These APIs opened the path to developer-friendly GPGPU, taking this paradigm out of the research laboratories and putting it on cross-platform commercial products, while challenging the CPU's *status quo*, as it ceased to be responsible for all the general-purpose processing.

In 2008, NVIDIA acquired AGEIA Technologies, a manufacturer of physics acceleration hardware and software, very similar in concept to GPGPU [152]. Also, NVIDIA announced full support for the OpenCL 1.0 specification [153].

In the same year, Intel announced the Larrabee microprocessor architecture [154]. This architecture aimed to merge a multi-core CPU and wide SIMD units, resembling GPGPU, into a single chip. However, this project was canceled, and there is currently no successor with a similar approach.

In late 2009, NVIDIA announced the new CUDA GPU architecture family, codenamed "Fermi", introducing IEEE 754-2008 floating-point standard, and ECC as the basis for GeForce, Quadro and Tesla GPUs, making the transition of algorithms to GPU much easier [155].

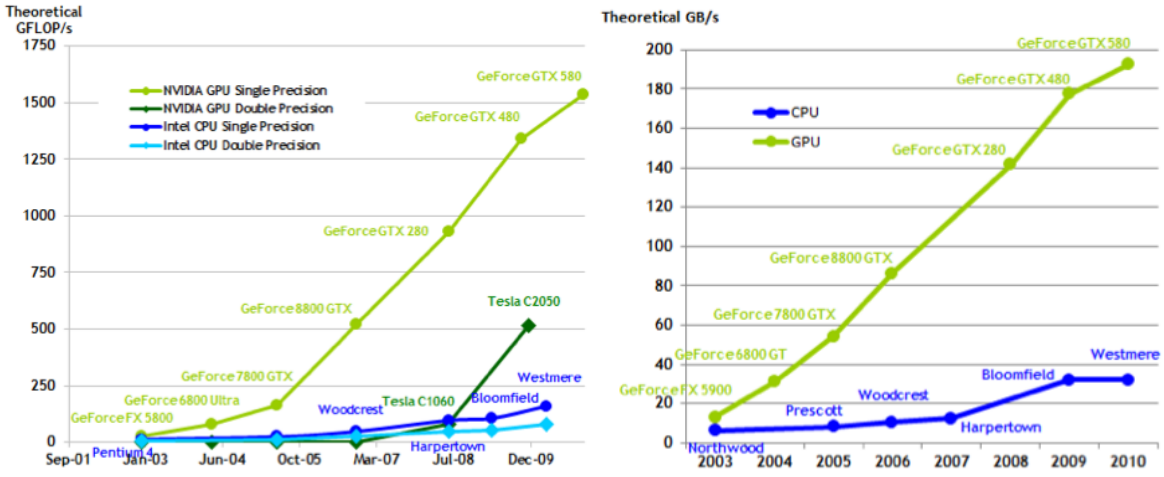
By January 2011, NVIDIA announced they have shipped their one billionth GeForce GPU [156]. GPUs are now present in personal computers, workstations, mainframes, game consoles, mobile phones, tablets, and embedded systems. On the same month, Intel agreed to pay $1.5 billion to NVIDIA for a cross-licensing agreement. Although this certainly not a manifesto for a close-knit relationship between Intel and NVIDIA, this agreement will likely contribute to the flexibility of parallel systems.

Mobile devices are now becoming multi-core. Basic mobile phones have a dual processor architecture: a micro-controller or general-purpose processor (e.g.: ARM) for the user interface and other basic tasks, and a DSP for speech codecs, channel codecs, Viterbi equalization. Now, with the amount of functionality present in mobile phones and tablets, such as the management of multiple transceivers (GSM, WLAN, Bluetooth, etc.), high resolution photography and video, Web browsing, 3D graphics and gaming, the industry is moving to multi-core, low-power SoCs [157]. OpenCL is also making its way to mobile GPUs. For example, the Imagination Technologies' PowerVR SGX and SGXMP series, included in many SoCs present in mobile devices such as Apple's iPhone and iPad, or Samsung's Galaxy S and Galaxy Tab, support it, although the software doesn't yet.

NVIDIA has an ongoing research project, Echelon, targeting energy efficiency, memory bandwidth, and programmability. The goal is to develop by 2017 an architecture combining a CPU and a GPU, with a double-precision throughput of 16 TFLOPS, 1.6 TB/s of memory bandwidth, all under 150 W of power [158, 159].

Virtually all personal computers in recent years have some kind of GPU in them. This is a great advantage over devices such as FPGAs. When compared to FPGAs, GPGPU benefits from a set of universal standards, APIs and tools, allowing shorter development times [160]. Furthermore, the price of GPUs is relatively low, in the order of hundreds of Euros, and they deliver hundreds of cores.

The latest NVIDIA GPUs can achieve nearly one teraflop of processing power, about an order of magnitude higher than CPUs, and a ten fold higher memory bandwidth [161]. Three out of the top five world's supercomputers employ GPUs [162]. Figure 3.1 shows the CPU



(a) Floating-point performance trends (from [163])

(b) Memory bandwidth trends (from [163])

Figure 3.1: CPU vs GPU trends

and the GPU floating-point performance and memory bandwidth trends.

The next step in the industry seems to be hybrid platforms, merging the CPU and the GPU in a single chip, eliminating the PCIe bottleneck (will be discussed in this text) [164]. AMD ships a low-end processor family known as Accelerated Processing Unit (APU), which consists of a CPU core and an OpenCL-compliant HD6000-based GPU. Intel also has a hybrid CPU-GPU platform, known as Sandy Bridge. However, this platform is not GPGPU-compatible. Still, it shows Intel's intent to merge both devices, and products similar to Larrabee will certainly be released in the near future, in a move similar to when processors started to include FPUs decades ago. There is also an emerging trend to merge the cloud computing and GPGPU worlds, as demonstrated by the Amazon EC2 GPU instances. The heyday of the CPU as the only general-purpose processing entity in a computer system is over.

GPGPU has applications in, among many other areas, n-body simulation, molecular modeling, computational finance, oil/gas reservoir simulation, medical imaging, computational fluid dynamics, environmental science, graphics (raytracing), cryptography, computer vision, product design, data analysis, technical computing and game physics [165, 63, 166, 167, 168].

3.2 GPU Architecture

Modern GPU architectures are a result of years of evolution in computer graphics. The algorithms in computer graphics are inherently parallel (per pixel, per vertex, per triangle, etc.). In order to generate an image, the GPU pipeline takes a set of polygons as input, and outputs a set of pixels to the framebuffer. As the polygons and pixels are independent of each other, they can be processed by parallel units (data parallelism). The computation is split into several stages, a **pipeline**, that can be overlapped. Until the beginning of the 2000s GPUs this pipeline wasn't programmable, and was named **fixed-function pipeline** [169, 170, 171].

With the introduction of vertex and fragment shader processors, due to increasing demand

of computer graphics power and flexibility, the **programmable pipeline** appeared. Note, however, that most pipeline stages remained fixed-function. Programmability using fixed-function GPU pipelines could be achieved by employing multi-pass rendering techniques, but with a huge performance penalty when compared to the programmability in hardware [172].

GPUs present a very predictable memory access behavior. Computer graphics often use texture mapping to make objects more realistic. Textures are composed by texels (abbreviation of texture elements, or texture pixels). When a texel is read, after a few cycles, the neighboring texel will be read. The same applies for pixels written. An opportunity to optimize memory organization emerges [170].

General-Purpose computation on Graphics Processing Units (GPGPU) is an implementation of **stream processing**, a paradigm related to SIMD, studied previously in Chapter 1. Given a set of 32-bit integer or floating-point data of the same type, called **stream**, a series of operations (with a reasonably general-purpose instruction set), called **kernels**, are applied to each element in the stream. CUDA supports the gather and scatter operations, allowing any random location of the device (global) DRAM memory to be read or written, as in a regular general-purpose CPU. For primitive processing, the GPU follows a SPMD programming model, where the elements are processed in parallel using the same program [170]. The data-parallel GPU stream programming paradigm applies four basic operations [170]:

- **Gather/Scatter.** Write or read from a computed location in memory. In GPGPU, gathers use the texture subsystem, storing the data as textures (i.e., images), and addressing the data by computing the image coordinates and by performing a texture fetch. Scatters are accomplished by rebinding data for processing as vertices, either using vertex texture fetches, or render-to-vertex-buffers. With CUDA and OpenCL, this whole process is hidden from the programmer, and the memory is exposed using standard C constructs, such as pointers, arrays, or variables.

- **Map.** Apply an operation to every data element in a collection. In a sequential program, it is typically expressed as a `for` loop. Map is performed as a fragment program (also called pixel shader) invoked on a collection of pixels, where each element is a pixel. For each pixel, the fragment program fetches the element data from a texture at a location corresponding to the pixel's location, and after performing the operation, stores the result in the output pixel.

- **Reduce.** Reduce a collection of elements to a single element or value, by applying repeatedly a binary associative operation. When computing certain operations, such as sums, one would perform sums in parallel on an ever-shrinking set of elements, instead of looping over an array and performing a running sum.

- **Scan.** Also known as parallel-prefix-sum, scan is used as a building block for sorting and matrix algorithms, among other applications. It takes an array A of elements and returns an array B, with the same length, in which each element $B[i]$ is a reduction of the subarray $A[1 \ldots i]$.

GPGPU is not a panacea. GPUs focus on having more **Arithmetic Logic Units (ALUs)** and less control units, making them highly useful for data-parallel applications. Applications with a great amount of instruction branching or with data that is difficult to parallelize are not ideal candidates for a GPU implementation. Another issue is that GPU

parallel programming (as well as multicore CPU programming) is not as straightforward as programming for a single processor, and it requires someone who is experienced with the GPU architecture and its parallel programming model. Although NVIDIA's CUDA has simplified the usage of GPU for general-purpose tasks, a number of hardware and software constraints must still be taken into account, in order to obtain high performance GPU-enabled applications [170, 173]. Figure 3.2 illustrates the architectural differences between CPUs and GPUs.

Figure 3.2: Architectural differences between CPUs and GPUs (from [163])

As the GPU usually sits on a discrete device, the major bottleneck in GPGPU applications is the relatively low bandwidth of PCIe transfers between the CPU and the GPU. Systems such as Sony PlayStation 3 and Microsoft Xbox 360 employ a GPU-CPU bus with a much larger bandwidth [170]. As mentioned in the previous section, future heterogeneous computing systems will certainly merge the CPU and GPU functionalities, addressing the PCIe bottleneck issues. A recent architecture, AMD Fusion, allows for a six-fold cost reduction when transferring data, except when transferring small batches of data [168].

3.3 CUDA

Current NVIDIA GPUs are based on the **Compute Unified Device Architecture (CUDA) architecture** [174], a "blueprint" for GPUs that are able to perform traditional graphics-rendering tasks as well as general-purpose tasks (GPGPU), and backed up by a C-based language called CUDA C [165].

3.3.1 Architecture

The CUDA architecture is composed by a scalable array of SIMD **Streaming Multiprocessors (SMs)**, also called **Multiprocessors (MPs)**. Each SM has a group of **Stream Processors (SPs)**, or **CUDA Cores**, sharing registers and shared memory, receiving instructions from a single issue unit, and executing them in a SIMD fashion. Each core is fundamentally a floating-point MAC unit. Compute capability 1.x devices have 8 cores per multiprocessor, while 2.x have 32 to 48 cores per SM. Each SM contains a number of Special Functional Units (SFUs), responsible for executing transcendental functions [175, 163, 172]. Figure 3.3 shows the GeForce 8800 architecture.

CUDA GPUs use a variant of SIMD, marketed by NVIDIA as SIMT, in which a set of threads (currently 32), called **warp**, is executed in lockstep. Warps are the basic unit of thread scheduling on a CUDA SM, which can be seen as a SIMD unit, where each core is a

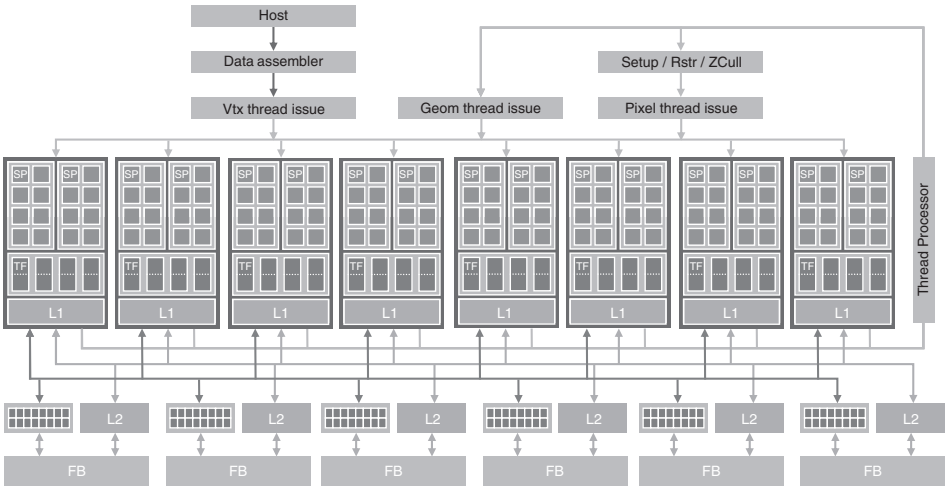


Figure 3.3: GeForce 8800 GTX and its unified programmable processor array (from [63])

SIMD element. The number of threads in a warp can vary with the architecture, and they are not part of the CUDA specification, although it is extremely useful to be familiar with this concept, in order to optimize kernels for a given architecture [172, 63]. When a warp terminates, a new one is launched on the vacant SM. Another important concept is that of a **half-warp**, the first or the second group of 16 threads in a warp [172, 63]. The more SPs in the GPU, the more threads can be run at a given time. CUDA has a complex memory hierarchy, which threads may access to. It will be studied in detail in Section 3.3.2.

CUDA threads execute on a physically separate **device** that can be seen as a coprocessor to the **host** running a C program. Host and device have their own memory spaces, **host memory** and **device memory**, respectively. The CUDA model separates the code that is to be executed on the host (CPU) from the kernels that are executed on the device (GPU). The code run by the host launches kernels to be executed on the device. Hence, this is called **heterogeneous programming**.

CUDA C programs are compiled with `nvcc` and consist of a mix of host code that runs on the CPU, and device code that runs on the GPU used to convert from the CUDA C to CUDA's Instruction Set Architecture, **Parallel Thread Execution (PTX)**. One can also write kernels with the ISA, without using the CUDA C language.

Multiple GPUs can be used, and the CUDA API allows the programmer to select the GPU he wants to issue commands to. Compute capability 2.x devices support peer-to-peer memory access, so that direct copies between devices can be made [163].

The **compute capability** of a device consists of a major and a minor revision numbers. The major revision number determines the core architecture, and the most recent is version 2, the Fermi architecture.

3.3.2 Programming Model

Programming a GPU is just another parallel programming application, and the theoretical speed-ups are, as usual, determined by Amdahl's Law or Gustafson's Law.

The workflow in CUDA is as follows: (1) send the data to the device to be operated upon; (2) perform computation on the data; and (3) send the results back to the host. Figure 3.4

illustrates this workflow.

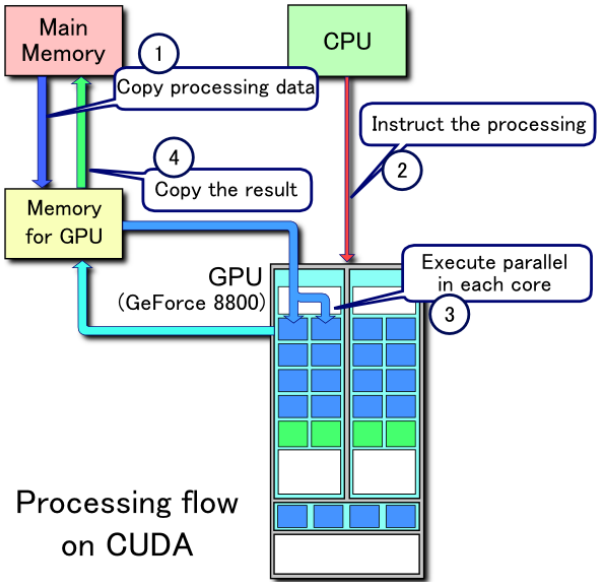


Figure 3.4: CUDA processing flow (from [176])

CUDA provides two C APIs: CUDA C Runtime API, and CUDA Driver API. Typically the programmer uses the higher-level Runtime API, which is built on top of the low-level Driver API, also accessible by the application [163]. Through the Driver API, the programmer has access to more features, such as CUDA contexts (resemble host processes), and modules (similar to dynamic libraries), but a lot of the hidden complexity in the Runtime API has now to be dealt with. Both APIs can interoperate with the Direct3D and OpenGL graphics APIs, which is useful for a great number of applications.

For best performance, CUDA requires a single CPU thread to operate all aspects of each GPU. This constraint has been relaxed in CUDA 4.0, but it is still regarded as the best practice. Multiple threads may create many different contexts with CUDA; however, performance will decrease as they contend with each other [177].

Kernels

CUDA C allows functions that are going to be run on the GPU to be specified. These functions are called **kernels**, and they are executed in parallel by different CUDA threads, on different data elements. A CUDA program is then composed by a serial CPU part and one or more kernels that are executed in parallel on the GPU by many threads.

Functions in a CUDA program can be of three types: **host functions**, the regular functions on the host side; **global kernels**, that are only callable by the host; and **device kernels**, that are only callable by other kernels on the device.

Only one kernel can be run on the device at a given time. However, compute capability 2.x devices can perform limited task parallelism, by executing a few kernels simultaneously, for a maximum of 16 [163].

Compute to Global Memory Access (CGMA) ratio is a metric defined as the number of floating-point operations performed for each access to the global memory within

a region of a CUDA program, and it should be as high as possible. The CGMA ratio has major implications on the performance of a CUDA kernel. It takes in the order of hundreds of cycles to fetch or store memory on the GPU, and only a few cycles to perform an add or multiply, thus it is vital to perform many computations in each kernel launch. For example, summing two vectors is probably faster using the CPU rather than the GPU, as there is a small number of operations to be made for each memory access [63, 163].

Instruction throughput is another performance issue to take into account. Arithmetic instructions with a high throughput should be used when possible. In compute capability 2.x (Fermi) architectures, double precision calculations cost twice as much as single precision calculations, not ten times as previously. GPU supports IEEE 754 floating point operations, and 24-bit integers have the same performance as floats [163]. Figure 3.5 details the throughput of arithmetic instructions.

	Compute Capability 1.x	Compute Capability 2.0	Compute Capability 2.1
32-bit floating-point add, multiply, multiply-add	8	32	48
64-bit floating-point add, multiply, multiply-add	1	16	4
32-bit integer add, logical operation	8	32	48
32-bit integer shift, compare	8	16	16
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	16	16
24-bit integer multiply (`__[u]mul24`)	8	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base-2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	2	4	8
Type conversions	8	16	16

Figure 3.5: Throughput of native arithmetic instructions (operations per clock cycle per multiprocessor) (from [163])

Thread Hierarchy

The programmer must first decompose the algorithm into threads. Threads should be made to minimize the communication between them, once again striving for the so-called embarrassingly parallel algorithms. If communication is necessary, shared memory should be used. However this memory is important to reduce redundant memory reads and improve performance [178].

As seen in Section 3.3.1, available processing units (cores) are scheduled the threads based

on resource availability. The highest performance in a warp is achieved when all the threads execute the same instruction. When the instructions are different, they run sequentially in separate branches and are executed independently. When threads in a warp disagree on branch paths, **branch divergence**, each branch path is executed serially, converging when all paths have executed, causing a severe impact on the performance of GPU kernels. This means that constructs like `if-then-else`, `for`, `while`, or `switch` should be avoided. Kernels with a large number of data-dependent control are unsuitable for the GPU [179, 172, 63, 180].

Research on solutions to reduce branch divergence is done by Han et al. [175] It proposes two software-based approaches: iteration delaying and branch distribution. **Iteration delaying** collects loop iterations that take a certain branch direction, and delays all others, so that only a direction is taken at a given time. **Branch distribution** reduces the divergent portion of a branch by factoring out structurally similar code from the branch paths. Tests were conducted by the authors, and they show an improvement of 60% to 80% on synthetic benchmarks respectively, and up to 12% and 16% respectively on real-world applications. These solutions are only usable on a very limited set of kernels, when all the branch paths can be determined before runtime.

Since `for` loops require branching, **loop unrolling** is often useful. Loop bodies may also access memories, and not have enough work to do. Unrolling the loop will frequently reduce memory accesses. GPGPU compilers often do an incomplete work unrolling loops, so a manual intervention is required [181, 182].

Threads are the fundamental means of parallel execution in CUDA. The launching of a CUDA kernel creates a grid of threads, so that all execute the same kernel function. They rely on unique coordinates to distinguish themselves from each other, and to correctly address the portion of the data being processed. Threads are organized into a two-level hierarchy using a block index (`blockIdx`) and a thread index (`threadIdx`), assigned by the CUDA runtime system, appearing to the kernel as built-in, preinitialized variables. Via these indices, along with the dimension of the grid (`gridDim`) and the dimension of each block (`blockDim`), the coordinates of the thread can be determined. A **grid** is organized as a 2D array of blocks. A **block** is organized into a 3D array of threads, and threads in a block can use shared memory to cooperate [165, 163].

Threads can be seen by the programmer in a 1-, 2-, or 3-dimensional view. For example, if the application is summing two vectors, probably the most useful view is one-dimensional. On the other hand, if the application is multiplying matrices, the bi-dimensional view is the most useful [173]. Figure 3.6 shows a 2D hierarchy of blocks and threads.

The developer must strive for the maximum GPU utilization. **Occupancy** is the percentage of available threads on a GPU working simultaneously, or the ratio of active warps per SM to the maximum number of warps. Current GPUs allow up to 768 and 1024 threads running simultaneously. Keeping the occupancy high helps hiding the latency of global memory accesses. The number of threads per block should be a multiple of 32 threads, so that optimal computing efficiency and coalescing are achieved. When choosing the block size, one has to pay attention to the fact that multiple concurrent blocks can reside on a MP, so the occupancy is not determined by the block size alone. For example, on a device with a maximum of 768 of threads per multiprocessor, using 512 threads per block will result in 66% occupancy, while using 256 threads per block will yield a 100% occupancy. Increasing the occupancy does not mean automatic performance gains, as there might be an overuse of shared resources (e.g., registers) by the threads in a block [180, 163]. The occupancy can be determined by compiling the CUDA code with a flag that outputs the resource usage (threads

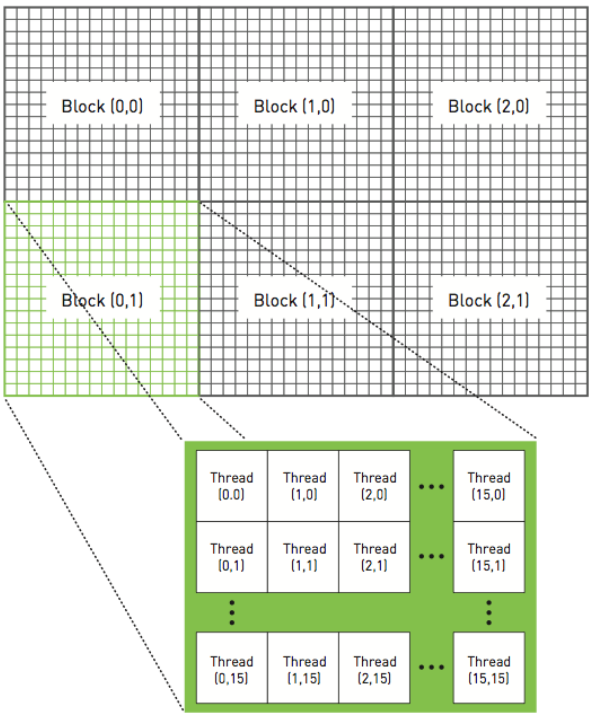


Figure 3.6: An example of a 2D hierarchy of blocks and threads (from [165])

per block, registers per thread, shared memory per block), and plugging in these values into the Occupancy Calculator spreadsheet, along with the compute capability of the device.

The CUDA Occupancy Calculator, a spreadsheet released by NVIDIA as part of the CUDA Toolkit, helps to compute the multiprocessor occupancy of a CUDA kernel. The user has to enter, for the kernel he is profiling, the number of threads per thread block, the registers used per thread, and the total shared memory used per thread block. The spreadsheet calculates the occupancy, the number of active threads, warps, and thread blocks per multiprocessor, and the maximum number of active blocks on the GPU. This information can then be used to tune up the kernel.

The developer can profile CUDA implementations using API mechanisms such as timers, and applications like NVIDIA Compute Visual Profiler. Any CPU timer library can be used, although one has to synchronize the CPU thread and GPU before starting and stopping the timer. CUDA has internal GPU timers, via **events** [163].

Memory Hierarchy

CUDA features a complex memory hierarchy, and threads have access to multiple memory spaces. One has to adapt the algorithms in order to take advantage of the different memory spaces available on the device. Memory management, as usual in the C language, is explicit, using functions inspired by the C language (`cudaMalloc`, `cudaFree`, `cudaMemcpy` and variants). CUDA devices are compliant with the IEEE 754-2008 standard, with some exceptions concerning rounding modes, floating-point exceptions and NaNs [163]. The following memories are available in CUDA, pictured in Figure 3.7:

Global memory, sometimes referred to as **device memory**, is the largest CUDA mem-

ory, but also the one with the highest latency, much higher than on register or constant memory. It usually stores the problem's data set. This memory is not cached, so accessing it (both when reading and writing) with the right pattern is crucial, called a **coalesced access**, which happens when threads in a half-warp access consecutive 4, 8, or 16-byte words. Put simply, the i-th thread must access the i-th word, or the amount of contiguous floating-point or integer numbers read must be equal to the number of threads in a half-warp. This process, although frequently tedious to accomplish, is vital for a performance increase, as the reads are batched into a single operation. When possible, on-chip cached memory (shared memory, registers), should also be used, to avoid unnecessary, slow accesses to the global memory. When accessing local or global memory, there are 400 to 600 clock cycles of memory latency, and can be hidden if there are sufficient floating-point operations (CGMA ratio). Global memory can be read/written by all threads [163, 180].

Constant memory is a 64 KB read-only memory shared by all MPs that allows for bandwidth conservation, compared to reading the same data from global memory, and near-register speed. When a thread of a half-warp performs a read from a constant memory address, the data can be broadcast to the other threads on the same half warp, saving up to 15 reads. Hence, only 1/16 of the memory traffic is generated. As this is a cached memory, a read from the constant memory costs one memory read from the device memory on a cache miss; otherwise, it just costs one read from the constant cache. Also, further half-warp reads to the same address will not incur in additional memory traffic. However, when the threads in a half-warp read different addresses, the reads get serialized linearly with the number of different addresses, making it often better to use global memory instead. Variables declared in constant memory space have the lifetime of an application [163, 165, 180].

Texture memory is a read-only cached on-chip memory for accessing the global memory, optimized for 2D spatial locality, shared by all MPs. Originally designed for texture units in Direct3D and OpenGL, as the texture access has a well-defined pattern, called **spatial locality**, where threads of a warp read nearby texture addresses, and therefore this process can be hardware-accelerated. The cache working set per MP for texture memory is currently between 6 and 8 KB. If the algorithm exhibits this access pattern like in texture mapping, this memory should be used for read-only data. [165, 180].

Shared memory is a cached on-chip memory with faster read/write access than global memory. This memory can not only be used by threads with the intent of communicating with other threads, but also to help to coalesce or eliminate redundant accesses to global memory. Its size is 16 KB for compute capability 1.x and 48 KB for 2.x, for each MP. It is divided into equally sized memory modules, or **banks**, allowing memory loads or stores of N addresses that span N distinct banks to be performed simultaneously. If the same bank is accessed by multiple threads, the accesses are serialized. There is an exception, when all threads of a half-warp access the same memory location, resulting in a broadcast, in compute capability 1.x devices. The mapping of threads to shared memory elements does not need to be one-to-one [165, 180]. Shared memory can be used to partition data into subsets called **tiles**, so that each tile fits into the shared memory. The computations on these tiles must be done independently of each other. The shared memory is much faster than the global memory, so, whenever the problem size and nature allows it (algorithm exhibits locality), accesses to global memory should be avoided by using the shared memory. Tiling is considered the most basic, and the most important optimization technique [183, 63].

Local memory is local to the thread scope-wise, but it is off-chip, stored in the global memory. And like global memory, it is also not cached on devices of compute capability 1.x,

making accesses to it as expensive as to global memory. Local memory is used by the `nvcc` compiler to hold local-scope automatic variables, when there is no register space available for a variable (each thread block is limited to 16 KB of register memory). Compute capability 1.x devices have 16 KB of local memory per thread, while 2.x devices have 512 KB [163, 180].

Each MP has a set of 32-bit **registers**. Compute capability 1.x devices have 8,000 to 16,000 registers per MP, while 2.x have 32,000 registers [163, 180].

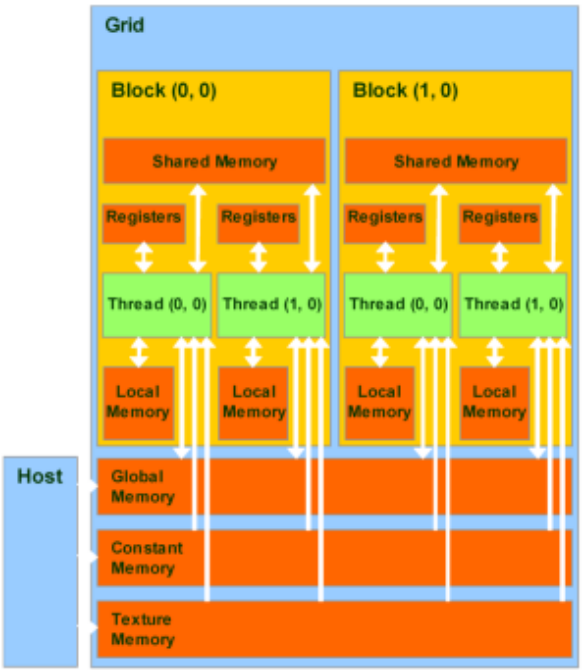


Figure 3.7: Memory hierarchy on the CUDA architecture (from [63])

3.3.3 Host Memory and Data Transfer

The GPU can be present as a discrete, dedicated device, communicating with the host via the PCIe bus, or as an integrated device, in which the host and the device share RAM memory.

Data transferred via the PCIe bus is stored in the device's global memory. Data transfer between the host and the devices should be kept to a minimum, as each transfer incurs in a great overhead, due to the relatively low bandwidth of the PCIe bus. Batching many small transfers into a single large transfer, and performing a large amount of computation on this data, is considered the best practice. Excessive host-device and device-host copies should be avoided as well. A good rule of thumb is that once on the device, the data should only be copied back after being fully processed [180].

There are three types of memory regions that can be allocated on the host memory.

Pageable memory is the regular memory type allocated by the `malloc` function call and variants. It can be swapped out to secondary storage [180].

Pinned memory, also known as **page-locked memory**, is a memory area that is never swapped out to secondary storage. With page-locked memory, since the buffer address is known, the GPU can apply DMA to copy data from or to the host. This type of memory is

useful when using a discrete GPU, in which data has to be transferred over PCIe. On devices with compute capability 2.x onwards, pinned memory enables concurrent kernel execution and copies between host and device memories, via **streams**, a mechanism that allows the management of concurrency. Page-locked host memory is allocated as **cacheable** by default. There is another mode, **write-combining**, in which the host's L1 and L2 caches are freed up for use with the rest of the application. Furthermore, this memory is not snooped (cache does not monitor address lines for accesses to memory locations that they have cached) during transfers across the PCIe bus, which can improve transfer speeds by up to 40%. Write-combining memory is useful for memory that the host only writes to, as reads are extremely slow. Page-locked memory can be allocated as **portable memory**, a block of host memory that can be used with any GPU on the system [180].

Zero copy memory is another type of memory, one that allows GPU threads to access host memory directly. This is also considered pinned memory, as it is never swapped out to secondary storage. The difference is that the CPU RAM is also the GPU RAM, so they have the same address space. This memory is useful when kernels read and write the data exactly once, even with discrete GPUs; and when using integrated GPUs, like those on laptop computers, as they share the RAM with the CPU [165, 180].

3.3.4 CUDA Toolkit and Other Utilities

A language based on ANSI C, designated **CUDA C**, is used to program for the NVIDIA GPUs. CUDA has bindings for the Fortran, Lua, IDL, Mathematica, MATLAB, .NET, Perl, Python, Ruby, Java, and Haskell languages.

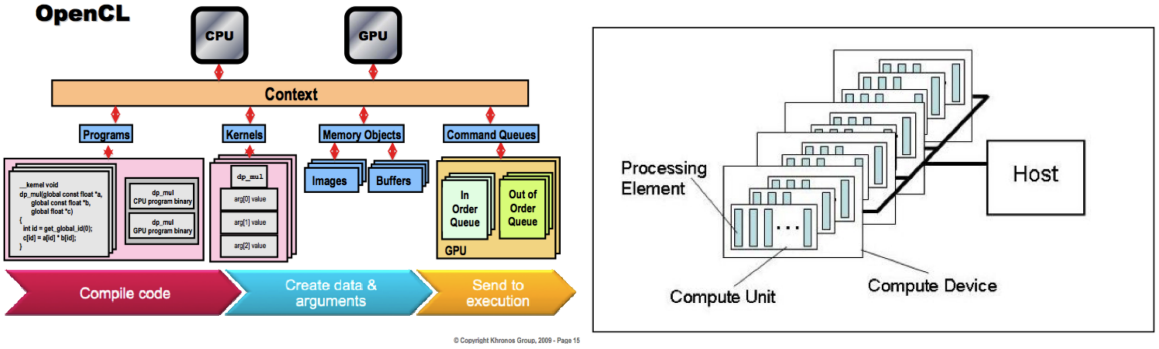
NVIDIA releases an SDK, CUDA Toolkit, for Windows, Linux, and Mac OS X. The current version is 4.1, released in January 2012. It contains the CUDA C compiler, `nvcc` (32- and 64-bit versions), as well as several utilities for debugging, CUDA and OpenCL samples, and architecture and API documentation [174]. On Linux/Mac OS X systems, `nvcc` will use GNU compiler, `gcc`, and on Windows systems, the Microsoft Visual Studio compiler, `cl` [184].

NVIDIA and third parties have released several libraries accelerated with CUDA. Some of the most popular are:

- **NPP.** NVIDIA Performance Primitives, a free library that includes functions for use with image, video, and other signal processing applications.
- **CUBLAS.** A CUDA version of BLAS, developed by NVIDIA. This is a library useful for performing basic linear algebra operations such as vectors and matrices operations. BLAS is usually part of LAPACK, a feature-rich linear algebra library.
- **CULA.** A linear algebra library, compatible with LAPACK, developed by EM Photonics in partnership with NVIDIA.
- **CUFFT.** An FFT library developed by NVIDIA. Can perform 1D, 2D, or 3D FFTs.
- **PhysX.** With the acquisition of AGEIA, their PhysX game physics middleware also runs on CUDA devices.

NVIDIA has several tools for assisting the CUDA development process:

- **CUDA-GDB.** A debugger similar to GDB, developed by NVIDIA, but for use with CUDA programs on Linux and Mac OS X [185].



(a) OpenCL model (from [192])

(b) OpenCL platform model (from [192])

Figure 3.8: OpenCL models

- **CUDA-MEMCHECK.** An error reporting tool for Linux, Windows, and Mac OS X, developed by NVIDIA [186].
- **NVIDIA Visual Profiler.** A Linux, Windows, and Mac OS X performance profiling tool for OpenCL and CUDA C/C++ [187].
- **NVIDIA Parallel Nsight** A tool for debugging, profiling and analyzing CUDA C, OpenCL, Direct3D, and OpenGL applications in Visual Studio [188].

3.4 OpenCL

Open Computing Language (OpenCL) is an open standard for cross-platform, heterogeneous parallel computing. It was originally proposed by Apple, and since June 2008 is maintained by Khronos [189], the same forum that supports OpenGL. The current version of the standard is 1.1, released in June 2010 [190].

Like CUDA, OpenCL is designed to allow parallel computing on heterogeneous devices, with the exception that the devices supported aren't limited to NVIDIA GPUs, but also include GPUs from other manufacturers, CPUs (including architectures other than x86), DSPs, and other generic accelerators. OpenCL-compliant device manufacturers include NVIDIA, AMD, S3, IBM, and Imagination Technologies.

The current OpenCL specification is very similar to CUDA, and most of the CUDA guidelines are valid in OpenCL, especially in NVIDIA devices [63].

3.4.1 Architecture

The OpenCL architecture is defined in four parts [191]: Platform Model, Execution Model, Memory Model, and Programming Model. Figure 3.8a shows an overview of the OpenCL architecture.

Platform Model

The OpenCL specification defines **platforms** for the interaction with OpenCL devices. These are implemented by the vendors (AMD, NVIDIA, etc.). This heterogeneous platform model consists of a **host** connected to one or more **OpenCL devices**. A device is divided into

one or more **Compute Unit (CU)**. Compute units are divided into one or more **Processing Element (PE)**, each with its own program counter. Computation on the device is performed on the PEs. Figure 3.8b shows OpenCL's platform model.

Execution Model

The host defines a **context**, or an environment, for managing OpenCL objects and resources. The resources can be of the following types:

- **Devices.** The OpenCL devices that will be used by the host.
- **Kernels.** The functions that will execute on the OpenCL devices.
- **Program Objects.** The program source that implements the kernels.
- **Memory Objects.** The data that will be operated by the programs. It is visible to the host and the OpenCL devices, and can be moved on and off devices. Objects can be classified as **buffers** or **images**.

Command queues are used to request operations to be performed by the device, like memory transfers, kernel executions, and synchronization. A command can be either synchronous or asynchronous. Each device has its own command queue.

Memory Model

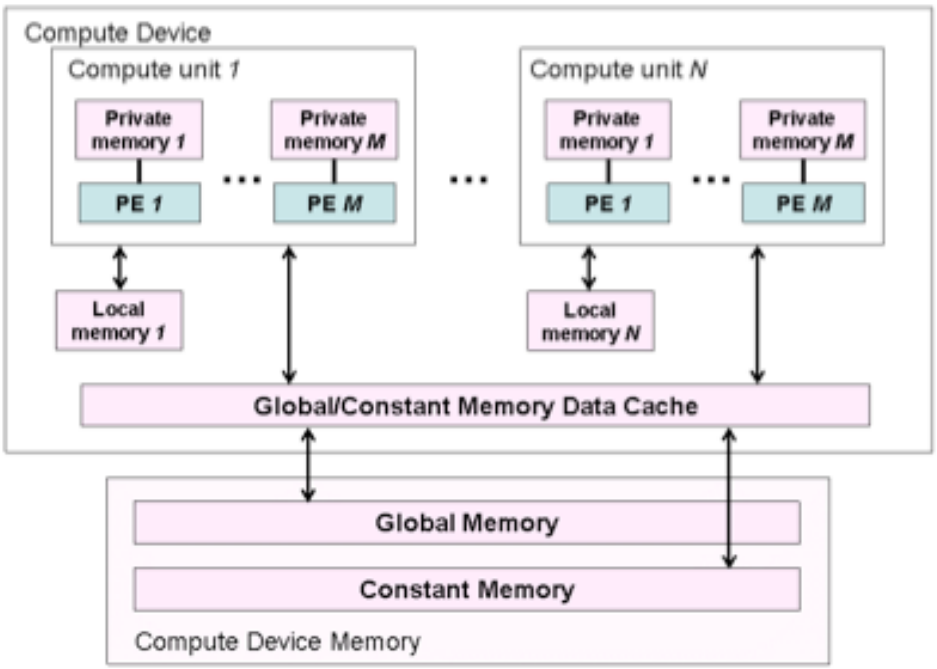


Figure 3.9: OpenCL device architecture. The host is not shown. (from [191])

Work-items executing a kernel have access to four distinct memory regions, as shown in Figure 3.9:

- **Global Memory.** All work-items in all work-groups have read/write access to memory object in this memory. There are no guarantees that this memory is cached, as it depends on the device. This memory can be dynamically allocated by the host, and supports read/write access by both host and devices.
- **Constant Memory.** This memory can be dynamically allocated by the host for the duration of a kernel (on CUDA it can only be statically allocated), and supports read-/write access by the host and read-only access by devices. Unlike CUDA, the size of this memory is not limited to 64 kB.
- **Local Memory.** Local memory is a memory region private to a work-group. It can be used to hold variables shared by all work-items in that work-group. It can also be used for tiling (hold regions of the global memory). This memory can be dynamically allocated by the host and statically allocated by the device. It can be accessed with read/write privileges by all work-items in a work-group (like CUDA's shared memory). The host cannot access it.
- **Private Memory.** It is private to a work-item, and cannot be seen by other work-items.

OpenCL	**CUDA**
Global Memory	Global Memory
Constant Memory	Constant Memory
Local Memory	Shared Memory
Private Memory	Local Memory

Table 3.1: Mapping of OpenCL memory types to CUDA memory types

Table 3.1 [63] shows the mapping between OpenCL and CUDA memories.

Programming Model

OpenCL supports both **data parallel** and **task parallel** programming models. The programming model also defines synchronization. Synchronization is possible between items in a single work-group, and between commands in a context command queue.

OpenCL	**CUDA**
Host Program	Host Program
NDRange (Index Space)	Grid
Work Item	Thread
Work Group	Block

Table 3.2: Mapping of OpenCL to CUDA data parallelism concepts

Table 3.2 [63] shows the mapping between OpenCL and CUDA data parallelism concepts.

3.5 Digital Signal Processing using GPUs

Contrary to DSPs, GPUs haven't been molded exclusively by DSP algorithms. They have been molded by graphics algorithms, which exhibit some similarities to signal processing

algorithms, such as the intensive usage of Multiply-Accumulate (MAC) operations.

Signal processing using GPUs is well documented, and it is certainly possible, as will be seen in the following sections. However, research on speech coding using GPUs is very scarce.

3.5.1 Filtering

Finite Impulse Response (FIR) and **Infinite Impulse Response (IIR)** (or recursive) filters are two of the most used constructs in signal processing. As mentioned in Chapter 2, these filters are very important in speech coding, and they benefit from the DSP dedicated MAC units.

FIR filters have the following form:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] = \sum_{i=0}^{N} b_i x[n-i].$$

The impulse response of the Nth-order filter is determined by the b_i coefficients, also known as tap weights. Each sample of the output signal $y[n]$ can be computed independently and the summation can also be parallelized, making the FIR filter a good candidate for a GPU implementation [193].

IIR filters are described by the following difference equation:

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-j]\right),$$

where P is the feedforward filter order, b_i are the feedforward filter coefficients, Q is the feedback filter order, and a_i are the feedback filter coefficients. Each sample of the output signal $y[n]$ not only depends on the input samples $x[n]$, but also on other pre-calculated output samples.

Research by Anwar et al. [193] achieved 3x to 40x speedups in IIR and 16-tap FIR filtering respectively, over the CPU versions. The IIR filters have a recursive, sequential nature, by having dependency on previously computed output samples. This problem was solved by decomposing the computation in two separate solutions, a process that will not be explained in this text.

Chang et al. [194] implemented Doppler signal processing on GPUs using CUDA, involving FIR and IIR filters. Although numerical errors due to the reduced floating-point precision are present, they arrived to the conclusion that using GPU for real-time ultrasound imaging is possible, given the transfer between the imaging system and the computer is fast enough.

Another class of filters, **Least Mean Squares (LMS)** filters, are widely used in digital signal processing, for tasks such as noise reduction, and echo cancellation. Bozejko et al. [195] study the implementation of the LMS algorithm using CUDA. This paper concludes that CUDA is a good fit for the LMS algorithm, because the most computationally intensive part of the algorithm, the process of updating the filter coefficients, can be reduced to matrix multiplications, which are easily parallelizable.

3.5.2 FFT and Other Transforms

Fast Fourier Transform (FFT) is a fast algorithm used to compute the Discrete Fourier Transform (DFT), a mathematical method that transfers data series from the time or space

domains to the frequency domain. The size N DFT is calculated for each number in the series, in the form:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk},$$

where k is an integer ranging from 0 to $N-1$. FFT implementations typically use a divide-and-conquer strategy to divide the computation into smaller DFTs. It is a very important construct in fields such as spectral analysis, signal processing (including audio), and data compression, and it is usually one of the most consuming tasks of an application [196].

Kocak et al. [197] discuss the applicability of CUDA for the FFT algorithm, with the intent of using it for increasing the throughput of wireless baseband processing in WirelessHD. The processing capabilities exceeded the required, decoding up to 4.36 Gbps. I/O limitations were encountered due to the PCIe limited bandwidth, and the solution was to reduce the data accuracy to 8-bit, instead of 32-bit data. Then, the data was expanded to 32-bit on the device and speed was improved by 27% versus the CUDA FFT libraries.

Gu et al. [196] criticize that GPU FFT performance numbers usually discard the transfer times between the host and the device, as performance is severely limited by the PCIe bandwidth. Besides, the FFT calculation on GPUs is limited by the relatively small device memory size. The proposed solution uses a decomposition framework based on the Cooley-Tukey algorithm that optimizes the CPU-GPU data transfers and the balance of on-GPU computation for 1-, 2-, or 3-D FFTs. It also introduces a blocked buffer technique for 1-D FFT data transfers. The end result is the ability to compute FFTs of a size much larger than the GPU memory.

Clemente et al. [198] research GPU usage in Synthetic Aperture Radar, an imaging radar for earth observation. Synthetic aperture radar processing requires advanced signal processing techniques and intense computational effort. The Doppler algorithm featured in this radar uses FFT. The performance tests show that, using a single NVIDIA Tesla C1060 GPU, the processing time is 15x lower than a CPU implementation running on an Intel Core 2 Duo E6850 at 3 GHz.

Fallen et al. [199] discuss the applicability of CUDA to the processing of radar data. Radar data processing requires a great amount of computational power, making heavy use of the FFT algorithm. Previously the required computations could not be done on-site in a timely manner, due to the lack of processing power, and also could not be uploaded to an off-site HPC center, due to the remote location of the laboratory. They conclude that both the GeForce and Tesla range NVIDIA GPUs are cost-effective and portable solutions to high-resolution radar data processing tasks, and that a single GPU was enough to meet the real-time requirements.

The **Discrete Cosine Transform (DCT)** and **Inverse Discrete Cosine Transform (IDCT)** real-time signal processing algorithms are implemented by Mohanty et al. [200] on a GPU, to be used in IPTV broadcasting, using old-fashioned GPGPU with NVIDIA C for Graphics (Cg) pixel and vertex shaders. The DCT has the following form:

$$X_k = \sum_{n=0}^{N-1} x_n cos\Big(\frac{\pi}{N}\Big(n+\frac{1}{2}\Big)k\Big), \quad k = 0, \ldots, N-1, \quad N \in \mathbb{N}.$$

This means that the operation transforms each of the N real numbers $x_0, \ldots, x_{N-1}$ into N real numbers $X_0, \ldots, X_{N-1}$, in an independent fashion, making it a good candidate for a

GPU implementation. The same applies for the IDCT. Mohanty et al. [200] conclude that the GPU is an excellent candidate for data intensive signal processing algorithms, despite the PCIe bottleneck.

Ujaldon et al. [201] analyze the use of CUDA for a 2D Fast Wavelet Transform implementation, concluding that it is not only a good fit for this algorithm, but also a two orders of magnitude improvement over a quad-core CPU.

3.5.3 Image Processing

Image processing algorithms are typically processor-intensive and parallelizable, as it generally involves independent processing of a massive pixel or feature set, ideal candidates for SIMD-style GPU implementations. Furthermore, image processing algorithms work with large memory buffers and needs frequent accesses to them, using regular and sequential access patterns, row-major or column-major order, where the arrays are flattened onto linear memory. The algorithms require intensive floating-point and logical computations [76]. Many applications would benefit from real-time image processing, but this is usually not possible, due to the processing demands of these algorithms [202].

The feasibility of CUDA for image processing is demonstrated by Park et al. [76], in the 3D shape reconstruction, feature extraction, image compression, and computational photography domains; and by Zhang et al. [202] and Misiorek et al.[203] in Sobel edge detection and homomorphic filters.

3.5.4 Video Processing

Video processing GPGPU feasibility has been extensively studied, and the motion estimation in video compression seems to be the usual target for GPGPU implementation. Motion estimation removes temporal redundancy within video sequences, where a picture is divided into several small macroblocks, and determines a motion vector from similar content in adjacent pictures [204].

For the H.264 codec, x264-cuda [205], work by Rodriguez et al. [206], Huang et al. [204], and Marth et al. [207] (OpenCL), demonstrate the feasibility of GPGPU for this motion estimation process. The same procedure has been done for an MPEG-4 (Part 2) encoder by Ailawadi et al. [208], and Yang et al. [209].

There are a few commercial, closed applications on the video market that support CUDA, like Adobe Premiere Pro [210] commercial non-linear video editing software, for effect processing, scaling, deinterlacing, etc.; Badaboom 2.0 [211], an audio and video commercial converter for Windows; and CyberLink MediaEspresso 6.5 [212], an audio, video, and image transcoding commercial application for Windows XP, Vista and 7.

3.5.5 Audio Processing

Audio usually does not exhibit a high degree of inherent parallelism [88].

The only work on speech coding using GPUs is Goldmann's master's thesis [88], who implemented a TETRA encoder to determine the amount of channels that could be processed in real-time using CUDA. The TETRA codec is based on the CELP technique. The author states that the C reference code requires extensive modification in order to adapt it to a particular architecture, and the reference implementation provides a 16-bit fixed-point format.

Besides, there are functions for each arithmetic operation, to facilitate the adaption to DSP hardware, making the code extremely hard to read.

First, a naïve implementation, with no parallel processing, running on a single GPU thread, was made. The results were disappointing, as a three-second speech sample took 5.1 seconds to process, while on the CPU it took 0.14s, or 36x slower on the GPU, far from real-time, leading the author to question the purpose of the whole endeavor. This experiment proved the poor performance of GPUs for sequential work. The reasons for this poor sequential performance were analyzed:

- Since all signals reside in global memory, and the global memory latency is very high, complex calculations with frequent accesses to buffers are a performance sink.
- As GPUs don't have automatically managed caches (like CPUs), the usage of the shared memory must be programmed manually.
- Even if using shared memory correctly, complicated access patterns can cause bank conflicts and the serialization of access operations, and are not expected to be avoided.
- The MAC instructions found on the GPU were not found in compiled assembler output files, therefore demonstrating poor instruction mapping.

As audio is not inherently parallel, an effort had to be made to make the GPU process a large audio data set. Data parallelism was drawn from the simultaneous processing of independent speech channels.

The author concluded that a CELP coder is computationally intensive, with limited inherent data parallelism. The GPUs work better when given enough work, but some load can be gained by parallelizing the most complex encoder stages. After profiling the encoder, the author decided to implement the following operations: analysis filter, synthesis filter, backward filter, algebraic codebook search, gain quantization, the calculation of the autocorrelation matrix which is employed in the codebook search, and the split VQ of LSPs after conversion from LPC coefficients. This was accomplished by exploiting independent iterations in loops, i.e., by processing unrolled loops in parallel.

The results were disappointing, only being able to encode about 90 channels in real time, worst than the 120 channels that the CPU was able to handle. The author states that future architectures with better branch handling would be an advantage for speech coding. Furthermore, the author states that due to the tedious development process that is speech coding, performing quick experiments with new ideas is nearly impossible.

Gjermundsen [213] implemented, as part of a Master's thesis, the Speex Acoustic Echo Cancellation filter in OpenCL, so that it can run on both CPUs and GPUs. This filter is an optional part of the Speex codec's preprocessing module, and it "subtracts" the echo of the sound played through the speakers from the sound picked up by the microphone.

Acoustic echo cancellation uses adaptive filters, self-adjusting algorithms that try to approximate a real system, like room acoustics, microphones, and speakers, based on an input signal over time. The signal produced by the filter is then subtracted from the signal coming in from the microphone.

Multidelay Block Frequency Domain Adaptive Filter (MDF) is a frequency domain variant of the LMS filter, where an adaptive filter approximates an unknown desired filter, by minimizing the least mean squares of the error signal. A number of blocks (frames) of fixed size, designated echo tail, is transformed into the frequency domain using FFTs. Then, all the

blocks are multiplied with corresponding stored weights, and added together into one block. The block is transformed back to the time domain, using a 1-dimensional FFT. Half the points in the transformed block are output as the filtered frame, and the value is subtracted from the initial frame, to find the error value. The error frame is padded and transformed into the frequency domain using a real-to-complex FFT. Each weight is updated by adding them to the calculated least mean square of the transformed error frame.

Speex, in particular, uses a MDF variant called Alternatively Updated MDF (AUMDF), where a single weight is updated for each frame that is processed by the filter, instead of having separate weights for each frame of the echo tail, in regular MDF, reducing the amount of FFTs and IFFTs from 40 to 2.

The OpenCL implementation was made to fit into the Speex framework as seamlessly as possible. For each filter instance, the filter state, a large structure with 26 buffers used by the filter, for temporary values and values that persist between input frames, has to be kept. Initialization and cleanup of the state structure, the main filtering function and subroutines were adapted/rewritten in order to be performed on OpenCL buffers.

Speex is bundled with an echo cancellation test, and this implementation augments it by including the OpenCL version. It reads two files from disk, one with the reference audio, and one with audio containing echo, in the form of raw samples. Frames from both files are fed into the echo cancellation filter, which returns a single output frame, and is then written to an output file. Modifications were made so that it could accommodate multi-channel input/output, higher sample rates, and larger frame sizes.

The original Speex MDF implementation is sequential, so an adaptation was done so that the available hardware could be fully utilized. The filter loops through all the samples in the input frame or combined window (current and previous frame). This step is trivially parallelizable. Besides this parallelization, the outer loops, consisting of a number of input or output channels were also parallelized.

The results show that more computationally demanding configurations of the echo cancellation filter favored execution on GPUs. When more input and output channels are added, the execution time of the reference implementation increased steeply, and this is the situation where the massively parallel processing capabilities of GPUs can increase performance. When using 12 input and 12 output channels, 5.3x speedups were achieved; however, on less demanding input configurations, the GPU implementation suffered from the OpenCL overhead.

A non-realtime MP3 decoder implemented using CUDA is described in [214], speeding it up by 5x, by decoding frames in parallel. It analyzes the decoding process first with a DSP and CPU approach. MPEG-1 or MPEG-2 Audio Layer III (MP3) decoding is a process that comprises nine independent blocks, and in this solution the CPU splits the MP3 files into frames and transfers them to the GPU, where they are processed in parallel. Each GPU thread executes the same decoding stage on a different frame. This decoder only supports constant bit-rate, to guarantee that the frame size is always the same, due to the coalesced accesses. Since this is not realtime, it can exploit data parallelism within the audio file, and it decodes frames that are independent in parallel.

FlaCuda and FLACCL [215] are open source NVIDIA CUDA and OpenCL respectively implementations of the FLAC encoder, made using the .NET Framework. The latter features different GPU and CPU optimizations. It runs compute-intensive operations needed for FLAC encoding such as channel decorrelations, autocorrelation, and LPC including the Schur and Levinson-Durbin algorithms (matrix operations), for several frames at once. It claims 10x

faster compression when using a higher compression ratio on a GeForce GTX 285 GPU than on a Intel Core i7 940 CPU.

Chapter 4

Implementation and Tests

As seen in Chapter 1, the need for a PSTN/CS Gateway in IMS is of the utmost importance, as the access-independent provision of multimedia services depends on this element. In this chapter, the implementation of a software PSTN/CS Gateway using Asterisk will be discussed. The feasibility of Asterisk and codec GPU acceleration will be also be considered.

Throughout the elaboration of this dissertation, it became clear that, assuming that it is possible to use CUDA to speedup the transcoding process on a Media Gateway using Asterisk, three major interventions are required: (1) Build an Asterisk IMS-compliant MGW; (2) Change Asterisk threading model, in order to have only one thread per GPU; and (3) Implement codecs using CUDA, with the capacity to encode/decode frames from multiple channels.

4.1 Initial Proposal

Initially, by the dissertation proposal, it was suggested that the GPU-accelerated Media Gateway should use CUDA and the GStreamer framework. This would require the Media Gateway to be written from scratch, under the assumption that the use of CUDA would require little modifications to any existing code.

GStreamer [216] is a cross-platform multimedia framework, licensed under GNU LGPL, designed to facilitate applications that require audio and video playback, recording, streaming and editing. It is used in the GNOME desktop environment, several media players such as Songbird, and embedded devices like the Palm Pre and Nokia tablets with the Maemo operating system. It uses a pipelined design, in which a number of processing elements can be connected in order to obtain the desired behavior.

Based on the IMS requirements for controlling the Media Gateway, it had to implement an H.248/Megaco interface. Work on a H.248/Megaco daemon was started, as no open source implementations were found.

Work had as well begun on a CUDA version of the Speex speech coder, taking as a starting point its libspeex library, with the intent of adapting it to perform encoding/decoding on frames from several channels at the same time using the GPU.

For each Gateway channel, there would have to exist two GStreamer pipelines: one pipeline for sending audio from the Gateway to the terminal, with an instance of the encoder (PCM to the terminal's codec) and RTP elements; and another pipeline for receiving audio, with an instance of the decoder (terminal's codec to the internal PCM format) and RTP elements. $2N$

pipelines would exist for processing N channels, and each pipeline possibly with more than one thread. In other words, in a single conversation between two clients, if a client was using the A-law codec and the other was using G.722, four pipelines would have to be created. After gaining a deeper understanding of the literature on CUDA, it became clear that a dedicated CPU thread was required for all the frames being processed (explained later on), and that this approach would not work. In addition, there would be N instances of encoders and N instances of decoders, undermining the usage of a single encoder/decoder for all channels.

Whilst reviewing literature on existing software Media Gateways, Asterisk was found to be used in some IMS-related projects. The fact that Asterisk supports a wide range of speech codecs and VoIP protocols, while GStreamer only supports a few speech codecs (G.711 A-law/μ-law and Speex), also contributed to rethink the usage of this framework.

For these reasons, the GStreamer framework was abandoned as the multimedia foundation for the proposed Media Gateway, and Asterisk was adopted, for both its telephony and multimedia capabilities.

4.2 PSTN/CS Gateway

The presented solution proposes the implementation of the whole IMS PSTN/CS Gateway, instead of just the Media Gateway (MGW), following an approach similar to work by Reis [217]. As a reminder, the PSTN/CS Gateway is formed by the MGCF, SGW, and MGW nodes, coordinated with the BGCF. There aren't any open source implementations of the BGCF and the MGCF, and Asterisk does not implement an H.248/Megaco interface. Since the IMS specification allows several functions to be combined into a single node, Asterisk can be used to perform all these functions, along with telephony cards, thereby discarding their inner interfaces. Obviously, this can be a problem if there are any existing BGCFs, MGCFs, or SGWs, as they can't be utilized in this approach.

A few efforts have been made to integrate Asterisk and Open IMS Core, found in literature.

Wu and Aasgaard [218, 219] analyzed, in 2006, the migration of enterprise VoIP solutions (e.g., PBXs) towards IMS, by exploring the scenario where a user is registered to an IMS realm and wishes to access enterprise SIP solutions. They set two main objectives: (1) the user is required to register to the VoIP service from within an IMS realm; and (2) the user must successfully establish a call with another user by using the said VoIP service. They propose four solutions within three migration stages, according to the degree of adoption of IMS by operators. However they do not implement any of the solutions.

The first migration stage (Today), assumes there are few IMS operators and has one solution, named "Forking". It proposes that the IMS just forwards all enterprise traffic to the VoIP provider, in this case Asterisk.

The second migration stage (Someday), assumes there are some IMS-enabled operators, requiring more features, specifically the use of a presence server and more complex registration and session setup, and proposes two solutions: "Client based", and "Presence". The "Client based" solution uses an intelligent UE that is able to detect its presence in the IMS realm, and registers to the enterprise service dynamically. In the "Presence" solution, it is expected that the IMS operator and the VoIP enterprise service are able to share the user's presence, so that the operator can detect the user's presence and registers him automatically to the VoIP service.

The third, and final migration stage (Future), for use when all operators are IMS-enabled,

proposes one solution, dubbed "Link Registration". This solution requires the IMS operator to keep track of the user's link registration, using features that are not yet implemented by access technologies, so that VoIP enterprise services can be delivered to the UE, in a seamless, automatic way.

Yao [220] tries to implement the second solution presented by Wu and Aasgaard [218], dubbed "Client Based", and presents the problems that have arisen, using Open IMS Core as an IMS operator, and Asterisk as a VoIP enterprise service provider. Although he was able to successfully accomplish the client registration part of the solution, he wasn't able to enable the delivery of the enterprise services to the client's UE. The client registration was implemented by duplicating the S-CSCF, where one S-CSCF processes normal IMS SIP requests, while the other acts as a proxy that forwards the enterprise SIP requests to Asterisk. This procedure, however, did not work for the call setup goal. Then, a redirection server was employed to forward the session requests to Asterisk, but the P-CSCF wasn't able to forward these requests to their enterprise destination. Due to time limitations, the cause of the problem was not discovered by the author.

The only attempts found in literature for a PSTN/CS Gateway using Asterisk are by Reis [217], in 2008, and by MacDonald et al. [11], in 2010.

Reis follows the single node approach. The BGCF is implemented as an Asterisk dialplan, with code changes, where a URI with a telephone number triggers a call to the PSTN or to another IMS where the breakout will occur. The MGCF is implemented by Asterisk, with no configurations whatsoever. The MGW is done with configurations in Asterisk's SIP channel driver, and in the dialplan. The SGW is automatically taken care of, as Asterisk has telephony interface card support. It is not clear how Reis solves the SIP incompatibility problem.

MacDonald et al. implement a PSTN/CS Gateway, in a solution that resembles the "Presence" solution by Wu and Aasgaard, shifting the intelligence from the UE to the IMS operator. The Asterisk-based Gateway is implemented as an AS.

The deviation from the SIP standard as it is used by IMS, hinders the conception of a Media Gateway, an apparently easy task. MacDonald et al. found that the P-CSCF problem pointed by Yao was, after all, due to the inability of Asterisk to process additional SIP headers introduced by Open IMS Core.

After packet analysis during the exchange of messages between Open IMS Core and Asterisk during call setup, they found that Asterisk was rejecting the IMS `INVITE` messages with the status `402 Bad extension (unsupported)`. This led to the suspicion that IMS users were using the wrong header extensions during calls, but it was ruled out after noticing that between IMS calls, this behavior could not be detected, and that the problem had to be on the Asterisk side.

`Received SIP INVITE with unsupported required extension: precondition` was added to Asterisk's log files every time the `402` message was generated. So, they made a Bash script in order to find, among the thousands of lines of code in Asterisk, the method that would generate such message. By checking the surrounding code, they came up with the conclusion that Asterisk verifies for supported SIP headers before proceeding with the call setup. As IMS introduces additional SIP headers, this was determined to be the failure source. By simply commenting out the code where SIP headers of incoming messages are checked, Asterisk was able reply to SIP requests from IMS users. In Asterisk 1.8.9.0 (MacDonald et al. used version 1.4.22), the lines to be commented are 22037 and 22042 in the `asterisk-1.8.9.0/channels/chan_sip.c` file. These changes are only a quick fix and the authors mention that a permanent solution should not be difficult to achieve, since it is a

matter of handling correctly the IMS SIP headers.

The configuration files were not released, but they state that, since IMS is now in charge of registration, instead of Asterisk, the user data must be repeated in Asterisk's configuration files. The procedure to configure Asterisk as an AS was also not explained.

In short, MacDonald et al. concluded that Asterisk can be used within an IMS test bed to interface with the PSTN/CS network, as well as to access enterprise VoIP services. It should not be used for an enterprise development, as the problems with the SIP implementation were not fully solved, complicated configurations were added, and SS7 compatibility was not tested.

Due to time and hardware constraints, and since MacDonald et al. and Reis didn't release full solutions, it wasn't possible to test and update their procedures.

Within the IMS domain, no more than a few call tests were done, after deploying Open IMS Core.

Using Asterisk, SIP and outgoing calls to the PSTN were tested. The SIP to PSTN calls were done using a Motorola Wildcard X100P FXO card connected to the building's internal telephone network, and another phone connected to the network. Due to the lack of a FXS card, configurations and testing for incoming calls were not done.

The bridge between Asterisk and Open IMS was not completed.

4.3 GPU Acceleration and Asterisk

As mentioned in Chapter 1, Asterisk is a heavily multi-threaded application. For each channel, Asterisk creates a thread. Through code inspection and call tests, it was discovered that each channel thread has two translator instances, in the encoding and the decoding directions. The test consisted in having three SIP users (registered in Asterisk's `sip.conf`): Alice, Bob, and Carol. Alice and Carol were allowed to only use the G.722 codec, while Bob was allowed to only use Speex. Alice and Bob were given extensions in Asterisk's dialplan, so that calls between them could be performed. Another extension was created, so that Carol could call it, but with Music on hold as the answer, with the intention of creating a one-legged call. The G.722 and Speex Asterisk modules were modified to display a message on its CLI whenever a translator was created. A **translator** is a generic term, in Asterisk, for a converter between the PCM raw format and a given speech coder format, in either direction (the same concept applies to video). The results were that Asterisk instantiated two G.722 encoders, one G.722 decoder, a Speex encoder, and one Speex decoder. Since Carol's use case was only half-duplex, she only needed one G.722 encoder. For N full-duplex channels using a given codec, there would be N encoders, N decoders, and N threads that would possibly interface with the GPU.

According to NVIDIA's CUDA Readiness Guide [177], it is possible, since CUDA 4.0, to use many CPU threads per GPU, by having the CPU threads setting the device to use. Having potentially thousands of CPU threads for each GPU would harm the performance, because each call to change the context to another CPU thread incurs in overheads. Hence, the threading model of Asterisk must be changed in order to support a GPU implementation.

A proper threading model is one where a single CPU thread exists per GPU and handles all the CUDA operations with it, gathering a large amount of frames for processing (batching). In addition, some sort of scheduling policy must be established, so that translations from the same source format to the same destination format, from several channels, can be processed

together. A solution that was briefly considered was to make a kernel module, like those on DAHDI transcoding cards, to handle the GPU transcoding, but it was readily abandoned as there is no access to CUDA in the kernel space.

Currently, in the web server world, and motivated by competitions such as C10K [221], which challenges web servers to support at least ten thousand simultaneous requests, an abandonment of the archetypal heavily-threaded architectures is occurring. In a situation where there are hundreds or thousands of requests per second, context switching between threads becomes prohibitively expensive. Non-blocking, event-driven servers such as Tornado [222], lighttpd [223], or nginx [224] serve a large number of clients with a single thread. A similar approach in Asterisk would be very useful, since with a large number of simultaneous calls there can be hundreds or thousands of threads, and context switching is harming the performance. This approach would have the added advantage of having only one thread managing all requests to the GPU.

In general, applications fall into one of the two models [225]: synchronous I/O with a single connection per thread, where there is high concurrency and potentially thousands of threads, making this model difficult to scale; and asynchronous I/O with a single thread, where the application handles context switching between clients. The literature on event-driven applications is very scarce.

The optimal number of threads in an application should be approximately the number of CPU cores, and thread pools should be used so that the thread creation/destruction overheads are minimized [225].

The task of modifying the threading model is not a trivial one, and during the elaboration of this dissertation it was tried with no success, as Asterisk has a huge C codebase built in more than a decade, and most modules rely on its heavily multi-threaded model. It would require a major rewrite in order to support this threading model. Furthermore, channel drivers (SIP, MGCP, etc.) also have associated threads. Asterisk, as of version 1.8.9.0, has around half a million lines of C code, with poor documentation for developers. The use of threads allow the code to be much simpler, as each channel can be treated as an independent functional unit.

Another solution that was tried, was to leave Asterisk heavily multi-threaded, but having the threads dump their frames into a large buffer and wait for translation, translation that would be done by an extra thread handling the GPU tasks. No suitable (not to mention scalable) data structures were found that would meet this requirement. Besides, due to the abstractions in Asterisk, it is very difficult to understand its channel read/write mechanisms.

Introducing another method of communication, for example, having an external process communicating with Asterisk via Sockets or other IPC technologies, would bring a great amount of overhead. But again, there is not a simple way to modify Asterisk for this purpose.

The threading model change endeavor was eventually abandoned due to the increasing suspicion that the GPU codecs implementations would not be accelerated, rendering work on this module useless.

4.4 GPU Acceleration and Codecs

For all codecs, and within the context of the encoding/decoding of a single frame for a single channel, there is a negligible or even zero data parallelism to be exploited. As a comparison, in video transcoding, there is often the opportunity to exploit the data parallelism

intrinsic to compressing video in blocks, but not in speech compression. Besides, as this is a real-time application, and the latency requirements are very strict, there is not an opportunity to gather many frames belonging to the same channel for batched processing.

However, data parallelism can be created by processing several frames of different channels, at once, by encoding/decoding the ones that are using the same codec. Note that the CPU implementations only support encoding/decoding of one frame at a time. Each CUDA thread will perform the same operation (e.g.: signed-linear PCM to G.722, or G.729 to signed-linear PCM). This approach creates the added advantage of not needing inter-thread communication, making this an embarrasingly parallel problem, but, as shall be seen, there are other problems.

As seen in Chapter 3, one of the most important CUDA optimization strategies is to make sure all accesses to global memory are coalesced. Global memory typically holds the problem's dataset, in this case the speech frames that are going to be translated. They can be placed linearly in memory, but coalescing is not possible, as coders use input/output buffers with word sizes under 4 bytes (4, 8, or 16 byte word sizes are required). This problem can probably be solved through the use of the texture mechanism, but it will introduce unreasonable development difficulties. The throughput of arithmetic instructions involving integers is much lower than the throughput when using floating-point data, so this will be another problem that will emerge.

Based on the characteristics of speech codecs seen in Chapter 2, the potential suitability of CUDA will now be discussed. As seen in Chapter 2, speech coders characteristics can ultimately be narrowed down to arithmetic operations, quantization, filters, matrix operations, transforms, and codebooks.

Research shown in Chapter 3, shows that filters appear to be good candidates for CUDA implementations. The same goes for transforms and operations involving matrices. However, there are a few characteristics that are problematic in CUDA.

Arithmetic operations usually require the signal to be within certain limits, at the risk of losing signal-to-noise ratio catastrophically. The saturation operation always requires branching on general-purpose CPUs and GPUs, and this is a major performance hit on GPUs. Since all the frames being processed have absolutely no relationship with each other, as they belong to different channels, the results of operations will be unpredictable, so it is reasonable to assume that the worst case will happen, that all the possible branch paths will be taken. For the same reason, the branch reduction techniques mentioned in Chapter 3 are not useful. Sometimes it is possible to solve the branching problem by storing every possible value in a lookup table, but as shall be seen, there is not a good fit for lookup tables in CUDA architecture. DSPs with saturation arithmetic could do this without any branching involved and have no performance penalties. A test was done in which all saturation was removed from the G.722 codec on the CPU implementation. The negative effect on the voice quality was very perceivable, due to the loss of signal-to-noise ratio. Due to time constraints, no codecs with this mechanism were tested.

Most codecs use static and dynamic tables during their encoding/decoding processes, for operations such as quantization and codebooks. Static lookup tables are usually very large, and it is not practical to replicate them for each encoder/decoder instance, as it will have an impact on occupancy, so the best strategy is to share the tables among these encoders/decoders. Dynamic tables cannot be shared between encoders/decoders, as they are part of their internal states. It is not possible to use constant memory (only for static tables) to hold the tables, because the positions accessed are dependent on the values of the intermediate signals

during encoding/decoding, and so there is no guarantee that within a half-warp, all threads will read from the same address, as they ought to. Even if it were possible to use constant memory, its size is only 64 KB and read-only, making it too small to store tables for multiple codecs (tables could not be switched on demand). Texture memory cannot also be used for the tables for the same reasons, because texture memory requires threads to access nearby locations (2D locality). Global memory latency would be high if the tables were stored in it, as it has very slow access times, and it is useful only for transferring large amounts of data, in order to hide that latency. Shared memory is mostly useful to hold a subset of the dataset (tiling), and for communication between threads, but as Liu et al. [226] demonstrated, it can be used for storing lookup tables, as long as there aren't any bank conflicts (two threads try to access the same location within a bank). Again, due to the randomness associated with the input signals, it is not possible to guarantee that there won't be any bank conflicts. The impact of lookup tables will be tested in upcoming sections.

Internal coder state variables, used for several coding stages, require storage. In order to satisfy this requirement, a large number of registers will have to be used for each thread, reducing the occupancy. The reduction of occupancy will have an impact on the performance.

Many of the optimization techniques employed in DSPs for speech coders seen in Chapter 2 can be applied on CUDA, but the problems mentioned above overshadow any gains achieved through these optimizations.

4.5 Transcoding Benchmark

In order to confirm the idea that speech coder GPU acceleration is not attainable, an application was made. At first this test was integrated into Asterisk, for use with the then abandoned one-thread-per-GPU model, while reusing existing Asterisk codec code. Difficulties were found integrating CUDA code with the modular Asterisk architecture, but they were solved by compiling Asterisk with embedded modules. After the departure from this implementation, the code was made into a standalone application, and the codecs were extracted from Asterisk into the application.

The initial intention was to implement as many codecs as possible using CUDA, and to create a common framework, so that a set of GPU-accelerated routines could be applied to many different codecs. Due to the intrinsic complexity of speech coders, as even the "simplest" codecs are very complicated (as seen in Chapter 2), and after many months expended on this task, the simplest case was chosen to be tested, which is to test the G.711 codec only, focusing on the impact of lookup tables.

Work on test procedures that could support all the different codecs and their intricacies, in the form of different sampling rates, different input/output formats, etc. was done. The application and the G.711 CUDA implementation were ported directly to OpenCL, as the principles behind the usage of OpenCL with NVIDIA cards are the same. The result is a benchmarking application that is extensible to any other codecs, including codecs for non-speech purposes, with support for CPU, CUDA, and OpenCL implementations, with the ability to compare with a baseline test with a trivial kernel.

As all codecs studied require tables for some functionalities, and since G.711 is the only codec with a relatively straightforward implementation, it became obvious that it was the only speech coder that could be completed for this dissertation. The results can be generalized to other codecs, as they introduce additional tables and code complexity in the form of filters,

transforms, etc.

The application consists of two test sets: the first test set essentially determines if the host-GPU latency is good enough for speech transcoding, while the second shows the unsuitability of the GPU for speech coding. Each test set consists of a number of tests, which are run 20 times (configurable). Then, their execution times are averaged and written to a file. After running the tests on a machine, a Bash script calls a MATLAB script in order to generate the plots from the data. The set of axes used is always the same, to allow a better interpretation of the results.

The transcoding benchmark application was made in C (codecs) and C++ (tests), with the CMake build system, and was tested on Linux and Mac OS X systems. The project can be checked out at [227].

These tests were run on:

- Intel Core 2 Duo P8700 at 2.53 GHz, 4 GB RAM, NVIDIA GeForce 9400M (CUDA compute capability 1.1) with 256 MB of RAM, shared with main memory (integrated), running Mac OS X 10.7 Lion.
- Intel Core 2 Duo T7500 at 2.20 GHz, 2 GB RAM, NVIDIA GeForce 8400M GS (CUDA compute capability 1.1) with 128 MB of dedicated memory, running Ubuntu Linux 10.04 32-bit.

Due to an OpenCL bug on NVIDIA's implementation on Mac OS X Lion, it was disabled for this operating system.

4.5.1 Latency Test Set

This test set is done with the intent of testing the host-device latency and see if it meets the requirements for speech transcoding, when using a trivial kernel. The three classes of host memory were tested in CUDA: non-pinned (regular), pinned (page-locked), and zero copy, where available. When using OpenCL, there is no method to force the usage of pinned memory, as this is managed by the driver [228], so only the regular allocation case was tested.

The test begins by allocating host and GPU memory according to the number of "frames" to be processed. As the buffer filled with zeroes is copied to the device (except in zero copy mode), the CPU timer is started. Then, a trivial kernel that sums 1 to each element is run on the data, and the 256 threads per block organization allows for 100% occupancy. The kernel operates on floating-point data, instead of 16-bit integers as used in PCM for telephony, so that accesses to global memory can be coalesced. Tiling was not used in any of the tests. A copy from the device to the host is done (except in zero copy mode) and the CPU timer is stopped. The buffer is verified to check if the expected result was generated. Finally, the host and device allocated memory regions are deallocated.

Figures 4.1 and 4.2 show the latency test results for an integrated and a dedicated GPU, respectively.

As expected, pinned memory allows faster data transfers than non-pinned memory. Zero copy memory is faster than the other two strategies when using an integrated GPU; on the dedicated GPU example, as the data is read and written exactly once, zero copy memory poses an advantage over the other strategies. The OpenCL version cannot guarantee that it will use pinned memory, and zero copy memory is not available. Since on NVIDIA cards OpenCL is built on top of CUDA, the OpenCL version performs slower than any of the CUDA versions, due to the inherent overheads.

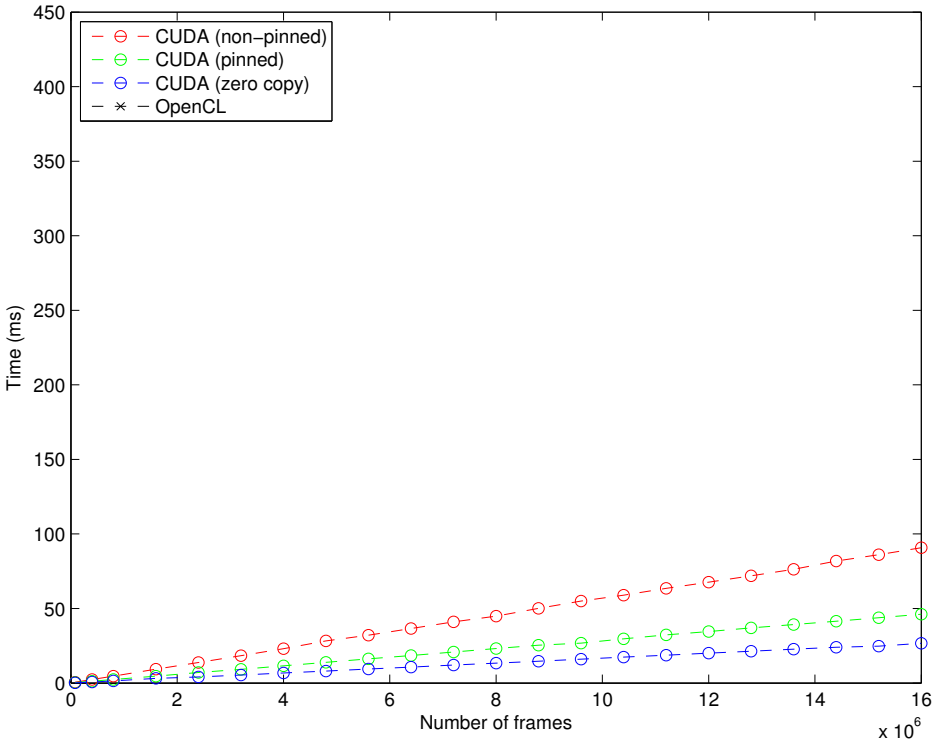


Figure 4.1: Latency test for different host memory allocation strategies on an integrated GPU (lower is better)

Although the kernel is very simple, with a very low CGMA ratio (only one addition), this test represents an upper bound when using CUDA and OpenCL, without using shared memory for optimization. In about 100 ms for the worst case (except with OpenCL), on both low-end integrated and dedicated GPUs, 16 million floating point elements can be copied to the device, processed using a trivial kernel, and copied back to the host memory. These results are encouraging, as the latency requirements of speech coding are met, and it shows that the host-to-device and device-to host PCIe transfer latencies are not a restriction, as long as, on the host side, a good scheduling policy is adopted.

4.5.2 Transcoding Test Set

In order to test the unfeasibility of GPUs for the task of speech transcoding, a second test set was created, so that CPU, CUDA, and OpenCL versions of codecs can be compared.

As mentioned before, only G.711 A-law was implemented. This coder does not have any computational tasks, besides performing lookups, after a shift right for the PCM to G.711 A-law case, for each sample to be encoded. Since all the studied coders use tables in many of their stages, the results can be generalized to the other coders. Besides, if this approach does not work in isolation, it will certainly not work in a multi-codec environment. A total of seven tests are made for the encoder and the decoder: a CPU version, and CUDA and OpenCL versions with the lookup tables in global memory, constant memory, and shared memory (local memory, in OpenCL).

The implementation takes as a starting point the G.711 A-law code in Asterisk. The

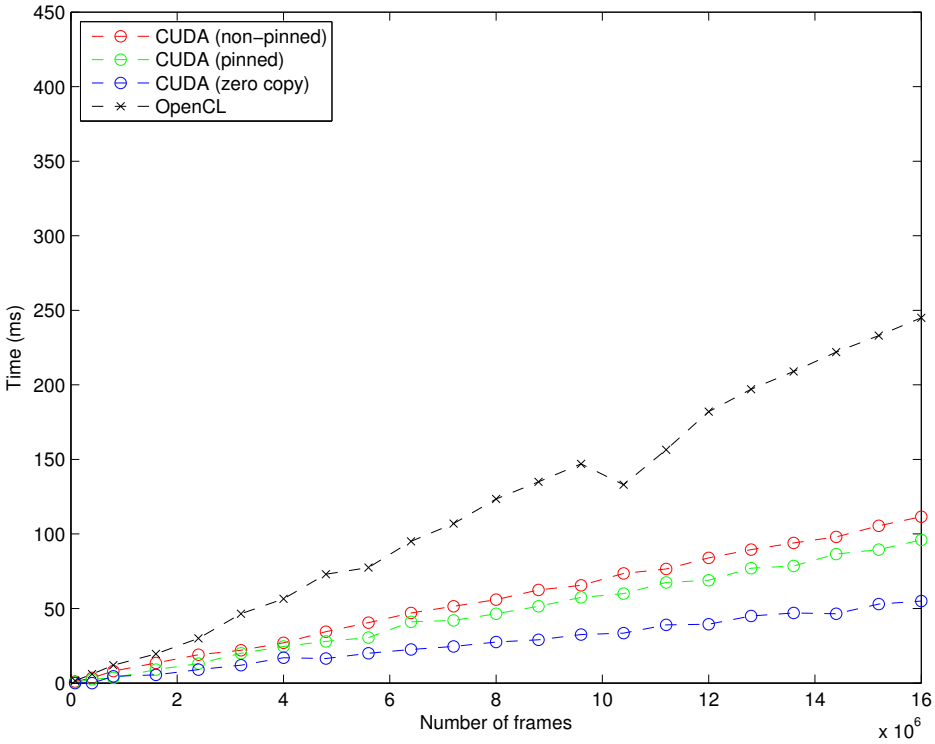


Figure 4.2: Latency test for different host memory allocation strategies on a dedicated GPU (lower is better)

---

existing CPU version first generates the 13-bit uniform PCM to A-law and A-law to uniform PCM lookup tables. For encoding (PCM to A-law), a macro is defined so that the sample to be encoded is right shifted in order to obtain the 13-bit PCM sample, and the output value is determined by accessing one of the lookup tables. For decoding, another macro is defined to output the PCM value, via lookup table, from an input A-law sample.

The CUDA and OpenCL implementations developed for this dissertation first generate the same lookup tables, but they are copied to global memory, constant memory, or shared memory, depending on the test. Instead of a macro to perform the lookup, this is done by a kernel, called for as many frames as there are in the test. In the case of shared memory (local memory, in OpenCL), since it has the lifetime of a kernel launch, the lookup table is pre-initialized during the kernel execution, from the global memory, and replicated for each multiprocessor. Since the number of threads in a block is smaller than the lookup table size, a given thread has to fetch more than one value from the global memory.

The frames being encoded belong to different channels, and are processed by different CUDA threads. The concept of channel in this test was adopted so that the first channel contains actual audio data, while the other channels contain random data, in order to create a test that is as realistic as possible. An interleaving scheme was adopted, as seen in Figure 4.3, where the frame from the first channel is followed by the first frame of the second channel, and so on.

In addition to the CUDA and OpenCL libraries, the SDL [229], SDL_mixer [230], libsndfile [231], and CppUnit [232] libraries were used. Using SOX [233], an 8 kHz audio file was generated from an audiobook, with the intent of loading 10 seconds of data.

Figure 4.3: Frame interleaving (C = channel; F = frame)

The non-realtime simulation starts by loading the wave file, in PCM format, and filling the first channel of the audio buffer. When testing the GPU implementations, the simulation starts with the voice data already on the device, ignoring the host-to-device and device-to-host latency. The objective is to isolate the effect of bandwidth on the GPU execution time, testing only the computational performance, and to see how the existence of lookup tables affects it.

Considering that 8,000 samples are equal to one second of audio, the test is run for 1 to 200 seconds of audio (configurable). On the CPU version, a simple loop goes through all the samples, and performs the translation. On the CUDA version, in order to maximize occupancy, the kernel which performs the lookup is called with 256 threads per block, as this is the optimal value, by the Occupancy Calculator. During this test, only one translator (an encoder or a decoder) is instanced, using a master thread that communicates exclusively with the GPU.

After a translation is performed, if so desired, the application can convert the data on the first channel back to PCM, to check the validity of the original translation. The data is then sent to an SDL_mixer channel, with the same sample rate and format as the original data, in order to be played back. For each test set, result files are generated for the CPU, CUDA and OpenCL (when available) versions.

PCM to G.711 A-law

The lookup table on this test consists of 8192 entries of one byte each (`unsigned char`).

According to the CUDA Occupancy Calculator, the occupancy for the global memory and constant memory cases is 100%, but for the shared memory case is only 33%, as the shared memory usage is very high, due to its use for holding the lookup table.

Figures 4.4 and 4.5 show the test results for integrated and dedicated GPUs, respectively.

On the integrated GPU case, the performance is comparable to that of the CPU, since the device (global) memory is the system RAM. However, on the dedicated GPU (with its own memory), the performance is much lower than on CPU, as the accesses are not coalesced, due to being random, and using one byte words. The constant memory implementation is not the slowest version, as it is a much faster memory. The shared memory (local memory, in OpenCL) implementations go completely off-scale, because each thread has to copy, using a `for` loop (which also introduces branching), lookup table data from the global memory, and the lookup table is relatively large.

The OpenCL version performs exactly like the CUDA version.

G.711 A-law to PCM

The lookup table on this test consists of 256 entries of two bytes (`short`), smaller than the PCM to G.711 A-law table.

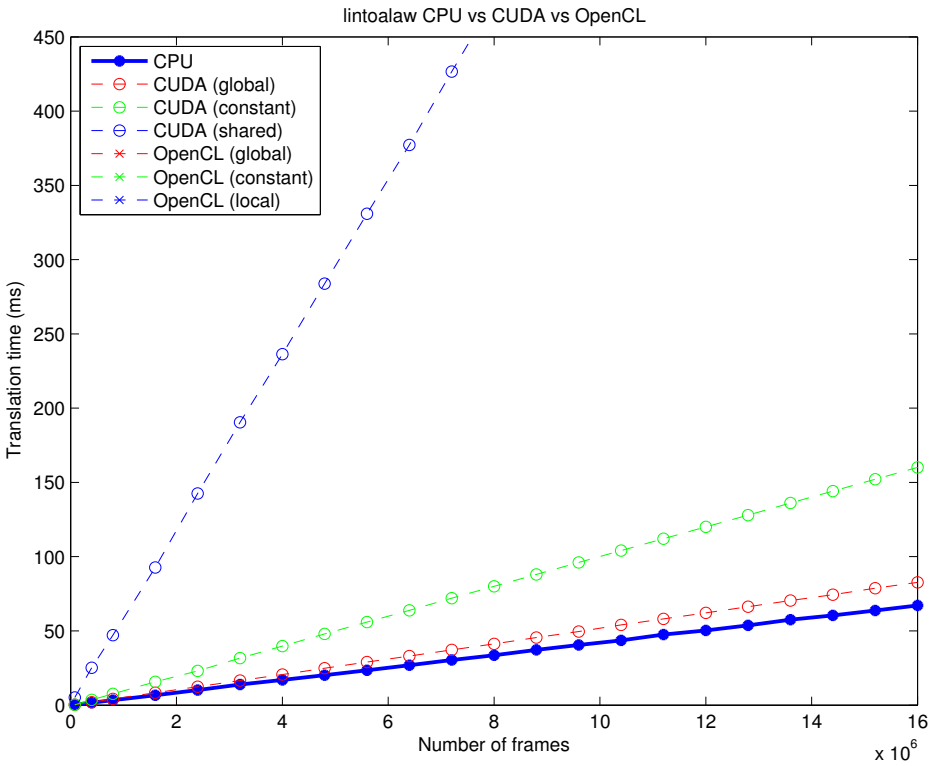


Figure 4.4: PCM to G.711 A-law on an integrated GPU (lower is better)

According to the CUDA Occupancy Calculator, the occupancy for all cases (global memory, constant memory, and shared memory) is 100%.

Figures 4.6 and 4.7 show the test results for integrated and dedicated GPUs, respectively.

Compared to the PCM to A-law example, as the lookup table is much smaller (256 entries versus 8192 entries), all lines are shifted downwards. The shared memory implementation does not perform as badly as in the other test, as the table that has to be copied to shared memory is not as large.

As before, the OpenCL implementation performs identically to the CUDA implementation.

Conclusions

GPU implementations of G.711 A-law perform consistently worse than their CPU counterparts, except when using small tables and integrated GPUs, which is not relevant, as the higher-end GPUs are never integrated. It is faster than the CPU version, but at several months development time (including familiarization with CUDA and OpenCL), and with a speedup of the same order of magnitude as the CPU version, it is certainly not worth the effort.

When compared to the Latency Test Set, even ignoring the data transfer times, it can be seen that the simple act of accessing memory in a large scale, as well as having non-coalesced accesses due to the usage of 16-bit integers, is very harmful to the performance. Due to an incompatibility of NVIDIA Visual Profiler with devices of compute capability 1.1, it wasn't possible to measure performance metrics in more detail.

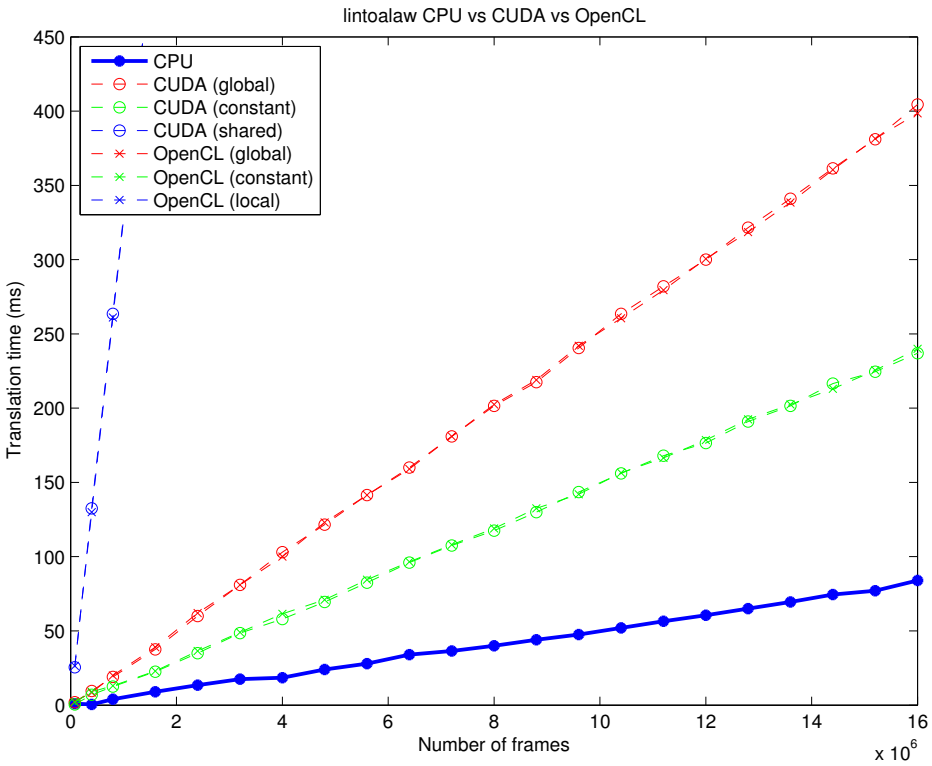


Figure 4.5: PCM to G.711 A-law on a dedicated GPU (lower is better)

This test concludes that lookup tables, required by most speech coders, are problematic in GPUs. It invalidates the implementation of the algorithms running 100% on GPU. It might be possible that some stages can be implemented on GPU, as long as those stages do not require any shared tables. Furthermore, saturation, integer performance, and coalescing issues, although not tested, hardly will make any performance gains possible. Other codecs will not allow such values of occupancy, as their storage needs are going to be higher, since encoder/decoder states have to be kept. As discussed before, it is not a good practice to perform many copies for the same data, so it will be very difficult to find a set of consecutive stages that can be successfully accelerated using the GPU. Along with the conclusion by Goldmann [88], this test suggests that, both for lower and higher complexity coders, GPU implementations may not pose any advantage, and as far as they were tested, they are actually slower than their CPU counterparts.

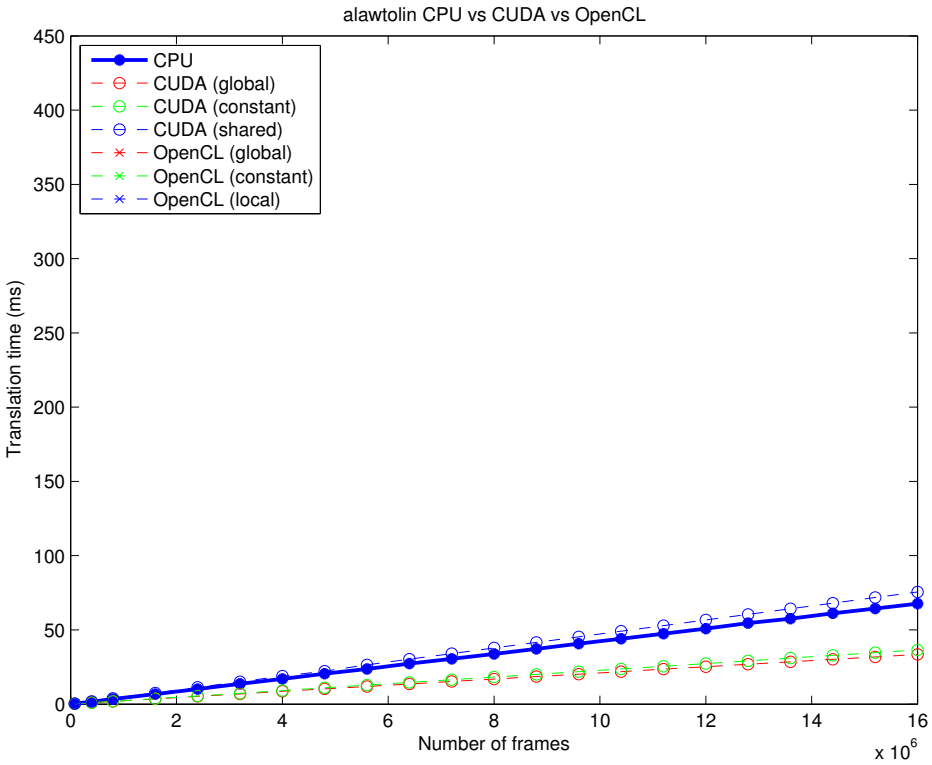


Figure 4.6: G.711 A-law to PCM on an integrated GPU (lower is better)

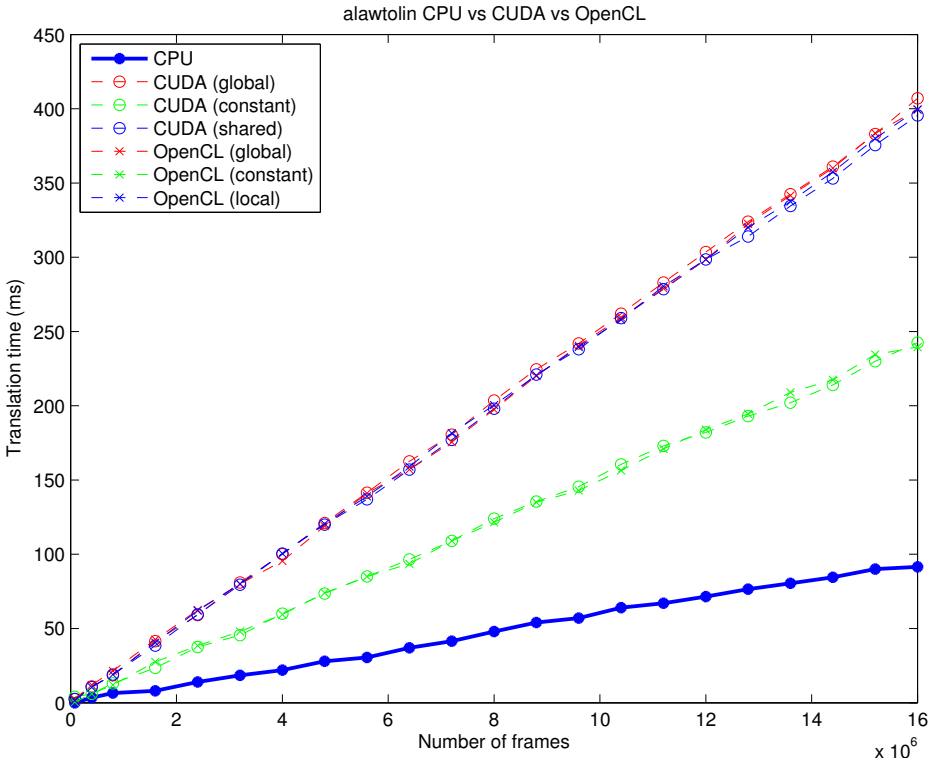


Figure 4.7: G.711 A-law to PCM on a dedicated GPU (lower is better)

Chapter 5

Conclusions and Future Work

This work demonstrated that, although it is possible to construct a limited-functionality software PSTN/CS Gateway using the Asterisk open source software, it is not worth using GPUs to assist the audio transcoding process.

The functionality required for transcoding is better suited for DSPs, as these have architectures molded throughout the years by digital signal processing algorithms, including speech coding. Research shows that digital signal processing is possible using CUDA, but speech coding in particular is a bad candidate.

This dissertation tested the usage of tables, as most codecs require them for operations such as quantization or codebooks, with unsatisfactory results for the G.711 A-law codec. The usage of integer arithmetic instead of floating-point arithmetic also harms the codec performance. Good candidates for GPU are algorithms with high arithmetic intensity involving floating-point data. In CUDA, a calculation is generally faster than a memory access. Other potential problems, such as arithmetic operations involving saturation, and storage required by the codecs, will hardly allow a speech codec to be accelerated using GPUs, but these were not tested due to time constraints.

Even if the GPU implementations were feasible, it would be a great development effort to modify Asterisk in order to support the batched approach, and to implement the speech coders, even taking general-purpose implementations as a starting point.

GPUs are expected to improve with respect in flexibility. When GPUs support massive task parallelism (i.e., threads executing different code), it is worth to revisit this problem, as different stages within a given coder can be processed in parallel. The book [63], co-written by NVIDIA's Chief Scientist, states that future CUDA devices will support task parallelism, making it possible to simultaneously execute several small kernels. With the Intel/NVIDIA cross-licensing agreement, CPUs and GPUs will certainly become much more flexible, and perhaps mitigate the issues that don't allow the transcoding process to be accelerated. Since hardware always adapts to the characteristics of the software, maybe in the future, depending on the demand, GPUs might have features like saturation arithmetic, or more flexible mechanisms for tables. If the opportunity of implementation arises, the transcoding benchmark application developed as part of this thesis can be used to test CPU, CUDA, and OpenCL implementations against each other.

There are other related applications worth exploring. Through code analysis, or with eventual future automatic methods, it would probably be possible to check for code sections in Asterisk with a high arithmetic intensity (not limited to speech transcoding), and offload

those computations to the GPU. Cryptography is another possible Media Gateway-related application on GPUs, as demonstrated by Liu et al. [226]; echo cancelation, another functionality present in MGWs is a good candidate for a GPU implementation, as seen in Chapter 3.

Bibliography

[1] G Camarillo and M A García-Martín. *The 3G IP multimedia subsystem (IMS): merging the Internet and the cellular worlds.* J. Wiley Sons, 2008.

[2] J V Meggelen, L Madsen, and J Smith. *Asterisk: the future of telephony.* O'Reilly Series. O'Reilly, 2007.

[3] D Merel, B Dempster, and D Gomillion. *Asterisk 1.6.* Packt Publishing Ltd., 2009.

[4] OpenVox. Asterisk SS7 Installation. Technical report, 2007.

[5] ITU-T. International telephone connections and circuits General Recommendations on the transmission quality for an entire international telephone connection.

[6] ITU-T. Packet-based multimedia communications systems. October 2009.

[7] Wikipedia. H.323. `http://en.wikipedia.org/wiki/H.323`.

[8] GNU Gatekeeper - a free VOIP Gatekeeper for H.323. `http://www.gnugk.org/`.

[9] Internet Engineering Task Force. SIP: Session Initiation Protocol. RFC 3261. *Internet Engineering Task Force*, 2002.

[10] James Wright. SIP: An Introduction.

[11] A MacDonald, R Cartas, and J Incera. Asterisk as a Public Switched Telephone Network Gateway for an IMS test bed. In *Telecommunications (ICT), 2010 IEEE 17th International Conference on*, pages 594–599, April 2010.

[12] C J Pavlovski. Service delivery platforms in practice [IP Multimedia Systems (IMS) Infrastructure and Services]. *Communications Magazine, IEEE*, 45(3):114–121, March 2007.

[13] Gilles Bertrand. The IP Multimedia Subsystem in Next Generation Networks. *Architecture*, 7(March), 2007.

[14] 3GPP. 3GPP TS 23.228 version 10.6.0 Release 10.

[15] M Jain and M Prokopi. The IMS 2.0 Service Architecture. In *Next Generation Mobile Applications, Services and Technologies, 2008. NGMAST '08. The Second International Conference on*, pages 3–9, 2008.

[16] P Podhradsky. New multimedia applications based on IMS NGN architecture. In *Systems, Signals and Image Processing (IWSSIP), 2011 18th International Conference on*, pages 1–4, June 2011.

[17] L N Saleem and S Mohan. An analysis of IP Multimedia Subsystems (IMS). In *Advanced Networks and Telecommunication Systems, 2007 First International Symposium on*, pages 1–2, 2007.

[18] M A Qadeer, A H Khan, J A Ansari, and S Waheed. IMS Network Architecture. In *Future Computer and Communication, 2009. ICFCC 2009. International Conference on*, pages 329–333, April 2009.

[19] Wikipedia. IP Multimedia Subsystem. `http://en.wikipedia.org/wiki/IP_Multimedia_Subsystem`.

[20] FOKUS. Open IMS Core. `http://www.openimscore.org/`.

[21] B Marsic, T Borosa, and S Pocuca. IMS to PSTN/CS interworking. In *Telecommunications, 2003. ConTEL 2003. Proceedings of the 7th International Conference on*, volume 2, pages 701 – 704 vol.2, June 2003.

[22] Internet Engineering Task Force. Megaco Protocol Version 1.0.

[23] Cisco Systems, Inc. `http://www.cisco.com`.

[24] Welcome to Nortel. `http://www.nortel.com/`.

[25] Avaya. `http://www.avaya.com/`.

[26] Dialogic. `http://www.dialogic.com`.

[27] Squire Technologies. `http://www.squire-technologies.co.uk/index.php`.

[28] Metaswitch Networks. `http://www.metaswitch.com/`.

[29] TelcoBridges. `http://www.telcobridges.com/`.

[30] Asterisk - The Open Source Telephony Projects. `http://www.asterisk.org`.

[31] FreeSWITCH. `http://www.freeswitch.org/`.

[32] B Jackson, J Brashars, and C Clark. *Asterisk Hacking*. Syngress, 2007.

[33] M A Qadeer and A Imran. Asterisk Voice Exchange: An Alternative to Conventional EPBX. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 652–656, 2008.

[34] A Imran, M A Qadeer, and M Khan. Asterisk VoIP private branch exchange. In *Multimedia, Signal Processing and Communication Technologies, 2009. IMPACT '09. International*, pages 217–220, March 2009.

[35] Asterisk developer's documentation. `http://www.asterisk.org/doxygen/trunk/index.html`.

[36] Amy Brown and Greg Wilson. *The Architecture of Open Source Applications*. 2011.

[37] F Iseki, Y Sato, and Moo Wan Kim. VoIP system based on Asterisk for enterprise network. In *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, pages 1284–1288, 2011.

[38] G. Thanos, A. Meliones, M. Marinidou, E. de La Fuente, and G. Konstantoulakis. *A 3GPP-SIP media gateway for the IP multimedia subsystem*. IEEE, 2007.

[39] Netfors Chan SS7 for Asterisk. `http://www.netfors.com/chan_ss7`.

[40] Digium. `http://www.digium.com/`.

[41] XORCOM. `http://www.xorcom.com/`.

[42] ATCOM. `http://www.atcom.cn/`.

[43] BroadTel. `http://www.broad-tel.com/`.

[44] DigiVoice. `http://digivoice.com.br/`.

[45] voicetronix. `http://www.voicetronix.com.au/`.

[46] Sirrix AG. `http://www.sirrix.com/`.

[47] Parabel. `http://www.parabel-labs.com/`.

[48] Nicherons International Inc. `http://www.nicherons.com/`.

[49] PhonicEQ, Inc. `http://phoniceq.com`.

[50] Rhino. `http://www.rhinoequipment.com/`.

[51] Sangoma. `http://sangoma.com/`.

[52] Elgato. `http://elgato.com.ua/`.

[53] Pika Technologies Inc,. `http://www.pikatechnologies.com/`.

[54] Synway. `http://www.synway.net`.

[55] OpenVox. `http://www.openvox.cn/`.

[56] Signalogic. `http://www.signalogic.com/`.

[57] Texas Instruments. TNETV3010 Infrastructure VOP Gateway Solution.

[58] Octasic. `http://www.octasic.com/`.

[59] Digium. `http://store.digium.com`.

[60] OpenVox Online Store. `http://store.openvox.cn/`.

[61] J M Rabaey, A P Chandrakasan, and B Nikolić. *Digital integrated circuits: a design perspective*. Prentice Hall electronics and VLSI series. Pearson Education, 2003.

[62] T Rauber and G Rünger. *Parallel Programming: For Multicore and Cluster Systems*. Springer, 2010.

[63] D Kirk, W M W Hwu, and W Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann. Morgan Kaufmann Publishers, 2010.

[64] J L Hennessy, D A Patterson, and D Goldberg. *Computer architecture: a quantitative approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 2003.

[65] J Eyre and J Bier. The evolution of DSP processors. *Signal Processing Magazine, IEEE*, 17(2):43–51, March 2000.

[66] Jorge Luis Ortega Arjona. *Architectural Patterns for Parallel Programming*. PhD thesis, University College London, 2006.

[67] Introduction to Parallel Computing. `https://computing.llnl.gov/tutorials/parallel_comp/`.

[68] Beowulf. `http://www.beowulf.org/`.

[69] SETI@home. `http://setiathome.berkeley.edu/`.

[70] Folding@home. `http://folding.stanford.edu/`.

[71] Apache Hadoop. `http://hadoop.apache.org/`.

[72] Jayanth Gummaraju and Mendel Rosenblum. Stream Processing in General-Purpose Processors. *Computer*, pages 94305–94305, 2004.

[73] Imagine. `http://cva.stanford.edu/projects/imagine/`.

[74] Merrimac - Stanford Streaming Supercomputer Project. `http://merrimac.stanford.edu/`.

[75] M Herlihy and N Shavit. *The art of multiprocessor programming*. Safari Books Online. Elsevier/Morgan Kaufmann, 2008.

[76] In Kyu Park, N Singhal, Man Hee Lee, Sungdae Cho, and C W Kim. Design and Performance Evaluation of Image Processing Algorithms on GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):91–104, 2011.

[77] David Salomon. *Data compression: the complete reference, Volume 10.* , 2007.

[78] A S Spanias. Speech coding: a tutorial review. *Proceedings of the IEEE*, 82(10):1541–1582, October 1994.

[79] W C Chu. *Speech coding algorithms: foundation and evolution of standardized coders*. J. Wiley, 2003.

[80] Jie Liang. ENSC 424 Multimedia Communications Engineering, 2005.

[81] Wikipedia. Psychoacoustics. `http://en.wikipedia.org/wiki/Psychoacoustics`.

[82] Wikipedia. Pulse-code modulation. http://en.wikipedia.org/wiki/Pulse-code_modulation.

[83] Multimedia.cx. PCM. http://wiki.multimedia.cx/index.php?title=PCM.

[84] ITU-T. Pulse Code Modulation (PCM) of Voice Frequencies. 1993.

[85] N M Sheikh, K I Siddiqui, M W Khan, S F Tajammul, and M Ashraf. Real-time implementation and optimization of ITU-T's G.729 speech codec running at 8 kbits/sec using CS-ACELP on TM-1000 VLIW DSP CPU. In *Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International*, pages 23–27, 2001.

[86] Lim Hong Swee. Implementation of G.729 on the TMS320C54x. Technical Report March, 2000.

[87] Antonio J Estepa, Juan M Vozmediano, Jorge López, and Rafael M Estepa. Impact of VoIP codecs on the energy consumption of portable devices. In *Proceedings of the 6th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, PM2HW2N '11, pages 123–130, New York, NY, USA, 2011. ACM.

[88] Axel Goldmann. *Towards GPU Speech Coding*. PhD thesis, Vienna University of Technology, 2011.

[89] Texas Instruments. http://www.ti.com/.

[90] Freescale Semiconductor. http://www.freescale.com.

[91] Analog Devices. http://www.analog.com.

[92] NXP Semiconductors. http://www.nxp.com/.

[93] W Patrick Hays. DSPs: Back to the Future. *Queue*, 2(1):42–51, March 2004.

[94] P M Embree. *C algorithms for real-time DSP*. Prentice Hall PTR, 1995.

[95] S A Suma and K S Gurumurthy. Evaluating the current market trends of DSP processors for speech processing applications. In *Signal Processing Systems (ICSPS), 2010 2nd International Conference on*, volume 1, pages V1–641 –V1–644, July 2010.

[96] A Z R Langi. Rapid development of a real-time speech coder on a TMS320C54x DSP. In *Electrical and Computer Engineering, 2002. IEEE CCECE 2002. Canadian Conference on*, volume 2, pages 1045 – 1048 vol.2, 2002.

[97] S Berger, Y Be'ery, B.-S. Ovadia, R Peretz, G Wertheizer, and Y Gross. An application specific DSP for speech applications. *Consumer Electronics, IEEE Transactions on*, 39(4):733–739, November 1993.

[98] P Wilson, J Puetz, A McCree, and D Wang. An integrated voice codec and echo canceller implemented in a single DSP processor. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '86.*, volume 11, pages 1333–1336, April 1986.

[99] G A Constantinides, P Y K Cheung, and W Luk. Synthesis of saturation arithmetic architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):334–354, July 2003.

[100] BORES Signal Processing. `http://www.bores.com/courses/intro/`.

[101] ITU-T. Narrow-band visual telephone systems and terminal equipment. 2004.

[102] Texas Instruments. Mu-Law and A-Law Companding Using the TMS320C2xx DSP. Technical Report October, 1997.

[103] FFmpeg. `http://ffmpeg.org/`.

[104] Ekiga. `http://ekiga.org/`.

[105] ITU-T. G.191: Software tools for speech and audio coding standardization. `http://www.itu.int/rec/T-REC-G.191/en`.

[106] Voip-info.org. voip-info.org - ITU G.711. `http://www.voip-info.org/wiki/view/ITU+G.711`.

[107] G.711 Protocol Overview. `http://www.lincoln.edu/math/rmyrick/ComputerNetworks/InetReference/127.htm`.

[108] ITU-T. 7 kHz Audio-Coding within 64 kbit/s. 1993.

[109] K Makelainen. Implementation of SB-ADPCM codec on a single TMS320C25 signal processor. In *Communications, 1990. ICC '90, Including Supercomm Technical Sessions. SUPERCOMM/ICC '90. Conference Record., IEEE International Conference on*, pages 1290 –1294 vol.4, April 1990.

[110] Avi Peretz and Cagdas Gumusoglu. *G.722 Wideband Speech Codec Implementation On BF-533 DSP*. PhD thesis.

[111] Y.-S. Lai and J Hartung. Implementation of the CCITT wideband coder using a high performance, fixed point DSP. In *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pages 1726 –1729 vol.3, April 1988.

[112] Steve Underwood. SpanDSP. `http://www.soft-switch.org/`.

[113] ITU-T. 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM). October 1990.

[114] Haiping Wang, Ju Liu, and Hongqing Miao. Implementation of bi-channel G.726 speech codec on fixed-point DSP. In *Neural Networks and Signal Processing, 2003. Proceedings of the 2003 International Conference on*, volume 2, pages 1635 – 1638 Vol.2, 2003.

[115] A Sharma and C P Ravikumar. Efficient implementation of ADPCM codec. In *VLSI Design, 2000. Thirteenth International Conference on*, pages 456–461, 2000.

[116] J Vehvilainen and J Nurmi. A processor core for 32 kbit/s G.726 ADPCM codecs. In *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on*, volume 3, pages 1932 –1935 vol.3, 1995.

[117] ITU-T. Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP). June 2007.

[118] Xiangping Kong, Hezhi Lin, Lianfen Huang, Jianan Lin, and Siyan Chen. Implementation of G.729 Codec Based on DaVinci Technology. In *MultiMedia and Information Technology, 2008. MMIT '08. International Conference on*, pages 11–14, 2008.

[119] M Banerjee, B A Vani, S Madhusudhan, and S Monga. Optimizations of ITU G.729 speech codec. In *Vehicular Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th*, volume 6, pages 3913 – 3918 Vol. 6, 2004.

[120] Cheng-Yu Yeh. Effective complexity reduction for G.729 standard. In *Electrical Engineering/Electronics Computer Telecommunications and Information Technology (ECTI-CON), 2010 International Conference on*, pages 1085–1088, May 2010.

[121] Ouyang Kun, Ouyang Qing, Li Zhitang, and Zhou Zhengda. Optimization and Implementation of Speech Codec Based on Symmetric Multi-processor Platform. In *Multimedia Information Networking and Security, 2009. MINES '09. International Conference on*, volume 2, pages 234–237, 2009.

[122] M Arora, N Lahane, and A Prakash. All assembly implementation of G. 729 Annex B speech codec on a fixed point DSP. In *IEEE International Conference on Acoustics Speech and Signal Processing*, volume 4, pages 3780–3783. IEEE; 1999, 2002.

[123] Jaewon Kim, Hyungjung Kim, Songin Choi, and Younggap You. The implementation of G.729 speech coder on a 16-bit DSP chip for the CDMA IMT-2000 system. *Consumer Electronics, IEEE Transactions on*, 45(2):443–448, May 1999.

[124] S Najafzadeh, M R Ghajar, A Nassery, and B Forouzandeh. Evaluation of Optimum Multi-Channel Implementation of CS-ACELP Codec on TMS320C6205. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 410–412, 2007.

[125] Geng Wang and Yongjie Zhang. The optimization of G.729 speech codec and implementation on the S3C2440. In *Electrical and Control Engineering (ICECE), 2011 International Conference on*, pages 451–454, 2011.

[126] ETSI. GSM 06.10 version 8.1.1 Release 1999.

[127] J Nurmi, V Eerola, E Ofner, A Gierlinger, J Jernej, T Karema, and T Raita-aho. A DSP core for speech coding applications. In *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, volume ii, pages II/429 –II/432 vol.2, April 1994.

[128] V Owall, P Andreani, L Brange, P Nilsson, A Wass, and M Torkelson. A GSM speech coder implemented on a customized processor architecture. In *Circuits and Systems, 1993., ISCAS '93, 1993 IEEE International Symposium on*, pages 235 –238 vol.1, May 1993.

[129] ETSI. GSM 06.20 version 8.0.1 Release 1999.

[130] N Lahane, V Agarwal, and S Sakri. All assembly implementation of GSM-HR speech codec on a fixed point DSP. In *TENCON 2004. 2004 IEEE Region 10 Conference*, volume A, pages 5 – 8 Vol. 1, 2004.

[131] Alan V Mccree and Thomas P Barnwell. A Mixed Excitation LPC Vocoder Model for Low Bit Rate Speech Coding. *Audio*, 3(4):242–250, 1995.

[132] Xiao Lin, Choon Boon Lim, Soo Peng Hoh, Gang Li, and Hai Bin Huang. Real time implementation of low bit rate speech encoder MELP on TMS320C54x DSP. In *Signal Processing Proceedings, 2000. WCCC-ICSP 2000. 5th International Conference on*, volume 2, pages 651 –654 vol.2, 2000.

[133] M Arora, P V S Babu, and M K Vinary. RISC processor based speech codec implementation for emerging mobile multimedia messaging solutions. In *Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on*, volume 2, pages 831 – 834 vol.2, 2002.

[134] Deock-Gu Jee and Song-In Choi. Real-time implementation of AMR-WB speech coder using TMS320C5509 DSP. In *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, volume 2, pages 1387–1390, 2005.

[135] R A Schilling and S L Harris. *Fundamentals of Digital Signal Processing Using MATLAB*. CL Engineering, 2011.

[136] M Akabri, B V Vahdat, and K Nayebi. Real-time implementation and optimization of ITU's G.728 on TMS320C64X DSP. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 896–901, 2005.

[137] R Nagal, M Kumar, and R Jain. Optimal real time DSP implementation of Extended Adaptive Multirate Wide Band (AMR-WB+) Speech Codec. In *TENCON 2008 - 2008 IEEE Region 10 Conference*, pages 1–6, 2008.

[138] Jie Yang, Sheng sheng Yu, and Mian Zhao. The implementation and optimization of AMR speech codec on DSP. In *Intelligent Signal Processing and Communication Systems, 2007. ISPACS 2007. International Symposium on*, pages 365–367, 2007.

[139] NVIDIA Corporation. NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256. `http://www.nvidia.com/object/IO_20020111_5424.html`.

[140] NVIDIA Corporation. NVIDIA Introduces GeForce3 for the PC. `http://www.nvidia.com/object/IO_20010530_5676.html`.

[141] NVIDIA Corporation. NVIDIA Introduces "Cg" – C For Graphics. `http://www.nvidia.com/object/IO_20020612_6724.html`.

[142] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[143] J A Kahle, M N Day, H P Hofstee, C R Johns, T R Maeurer, and D Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[144] Sh. http://libsh.org/.

[145] BrookGPU. http://graphics.stanford.edu/projects/brookgpu/.

[146] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM SIGGRAPH 2004 Papers*, 23(3):777–786, 2004.

[147] Michael D McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proceedings of the 2006 ACMIEEE conference on Supercomputing SC 06*, pages 11–17. ACM Press, 2006.

[148] Matthew Papakipos. The PeakStream Platform: High-Productivity Software Development for Multi-Core Processors.

[149] AMD's Close-to-the-Metal. http://sourceforge.net/projects/amdctm/.

[150] NVIDIA Corporation. New NVIDIA Products Transform the PC Into the Definitive Gaming Platform. http://www.nvidia.com/object/IO_37234.html.

[151] NVIDIA Corporation. NVIDIA Unveils CUDA-The GPU Computing Revolution Begins. http://www.nvidia.com/object/IO_37226.html.

[152] NVIDIA Corporation. NVIDIA completes Acquisition of AGEIA Technologies. http://www.nvidia.com/object/io_1202895129984.html.

[153] NVIDIA Corporation. NVIDIA Adds OpenCL To Its Industry Leading GPU Computing Toolkit. http://www.nvidia.com/object/io_1228825271885.html.

[154] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[155] NVIDIA Corporation. NVIDIA Unveils Next Generation CUDA GPU Architecture Codenamed Fermi. http://www.nvidia.com/object/io_1254288141829.html.

[156] NVIDIA Corporation. One Billionth GeForce Ships. http://blogs.nvidia.com/2011/01/one-billionth-geforce-ships/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+nvidiablog+%28The+NVIDIA+Blog%29.

[157] C H (Kees) van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[158] David Brooks. CPUs, GPUs, and Hybrid Computing. *Micro, IEEE*, 31(5):4–6, 2011.

[159] S W Keckler, W J Dally, B Khailany, M Garland, and D Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, 2011.

[160] Song Jun Park, D R Shires, and B J Henz. Coprocessor Computing with FPGA and GPU. In *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 366–370, July 2008.

[161] NVIDIA Corporation. GeForce GTX 590. `http://www.geforce.com/Hardware/GPUs/geforce-gtx-590`.

[162] TOP500 Supercomputing Sites. `http://www.top500.org/`.

[163] NVIDIA Corporation. CUDA C Programming Guide (version 4.1). Technical report, 2012.

[164] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[165] J Sanders and E Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[166] W W Hwu. *GPU Computing Gems Emerald Edition*. Applications of GPU Computing Series. Elsevier Science Technology, 2011.

[167] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[168] M Daga, A M Aji, and Wu-chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141–149, July 2011.

[169] Erik Wynters. Parallel processing on NVIDIA graphics processing units using CUDA. *J. Comput. Small Coll.*, 26(3):58–66, January 2011.

[170] J D Owens, M Houston, D Luebke, S Green, J E Stone, and J C Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[171] William Mark. Future Graphics Architectures. *Queue*, 6(2):54–64, March 2008.

[172] T M Aamodt. Architecting graphics processors for non-graphics compute acceleration. In *Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on*, pages 963–968, 2009.

[173] J F Croix and S P Khatri. Introduction to GPU programming for EDA. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 276–280, 2009.

[174] NVIDIA Corporation. CUDA Zone. `http://developer.nvidia.com/category/zone/cuda-zone`.

[175] Tianyi David Han and Tarek S Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1—-3:8, New York, NY, USA, 2011. ACM.

[176] Wikipedia. CUDA. http://en.wikipedia.org/wiki/CUDA.

[177] NVIDIA Corporation. CUDA Toolkit 4.0 Readiness For CUDA Applications. 2011.

[178] Patrik Goorts, Sammy Rogmans, Steven Vanden Eynde, and Philippe Bekaert. Practical examples of GPU computing optimization principles. In *Signal Processing and Multimedia Applications (SIGMAP), Proceedings of the 2010 International Conference on*, pages 46–49, July 2010.

[179] Wen-Mei Hwu, C Rodrigues, S Ryoo, and J Stratton. Compute Unified Device Architecture Application Suitability. *Computing in Science Engineering*, 11(3):16–26, 2009.

[180] NVIDIA Corporation. CUDA C Best Practices Guide. 2012.

[181] Damir A Jamsek. Designing and optimizing compute kernels on NVIDIA GPUs. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ASP-DAC '09, pages 224–229, Piscataway, NJ, USA, 2009. IEEE Press.

[182] G S Murthy, M Ravishankar, M M Baskaran, and P Sadayappan. Optimal loop unrolling for GPGPU programs. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, April 2010.

[183] Chang Xu, S R Kirk, and S Jenkins. Tiling for Performance Tuning on Different Models of GPUs. In *Information Science and Engineering (ISISE), 2009 Second International Symposium on*, pages 500–504, 2009.

[184] NVIDIA Corporation. The CUDA Compiler Driver NVCC. 2011.

[185] CUDA-GDB. http://developer.nvidia.com/cuda-gdb.

[186] CUDA-MEMCHECK. http://developer.nvidia.com/cuda-memcheck.

[187] NVIDIA Visual Profiler. http://developer.nvidia.com/nvidia-visual-profiler.

[188] NVIDIA Parallel Nsight. http://developer.nvidia.com/nvidia-parallel-nsight.

[189] Khronos Group. Khronos Launches Heterogeneous Computing Initiative. http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/.

[190] Khronos Group. Khronos Drives Momentum of Parallel Computing Standard with Release of OpenCL 1.1 Specification. http://www.khronos.org/news/press/releases/khronos-releases-final-webgl-1.0-specification.

[191] Khronos Group. OpenCL Specification. *ReVision*.

[192] David Kaeli and Benedict Gaster. AMD OpenCL University Kit, 2011.

[193] Sajid Anwar and Wonyong Sung. Digital Signal Processing Filtering with GPU. 2.

[194] Li-Wen Chang, Ke-Hsin Hsu, and Pai-Chi Li. GPU-based color Doppler ultrasound processing. In *Ultrasonics Symposium (IUS), 2009 IEEE International*, pages 1836–1839, 2009.

[195] W Bozejko, A Dobrucki, and M Walczynski. Parallelizing of digital signal processing with using GPU. In *Signal Processing Algorithms, Architectures, Arrangements, and Applications Conference Proceedings (SPA), 2010*, pages 29–33, 2010.

[196] Liang Gu, Jakob Siegel, and Xiaoming Li. Using GPUs to compute large out-of-card FFTs. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 255–264, New York, NY, USA, 2011. ACM.

[197] T Kocak and N Hinitt. Exploiting the Power of GPUs for Multi-gigabit Wireless Baseband Processing. In *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*, pages 56–62, July 2010.

[198] C Clemente, M di Bisceglie, M Di Santo, N Ranaldo, and M Spinelli. Processing of synthetic Aperture Radar data with GPGPU. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 309–314, 2009.

[199] C T Fallen, B V C Bellamy, G B Newby, and B J Watkins. GPU Performance Comparison for Accelerated Radar Data Processing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 84–92, July 2011.

[200] S P Mohanty. GPU-CPU multi-core for real-time signal processing. In *Consumer Electronics, 2009. ICCE '09. Digest of Technical Papers International Conference on*, pages 1–2, 2009.

[201] Manuel Ujaldon and Umit V Catalyurek. High-performance signal processing on emerging many-core architectures using CUDA. In *ICME'09: Proceedings of the 2009 IEEE international conference on Multimedia and Expo*, pages 1821–1824, Piscataway, NJ, USA, 2009. IEEE Press.

[202] Nan Zhang, Yun-shan Chen, and Jian-li Wang. Image parallel processing based on GPU. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 3, pages 367–370, March 2010.

[203] F Misiorek, A Dabrowski, and P Pawlowski. The ability to use parallel programing on graphics processor unit in digital signal processing. In *Signal Processing Algorithms, Architectures, Arrangements, and Applications (SPA), 2008*, pages 225–228, 2008.

[204] Yen-Lin Huang, Yun-Chung Shen, and Ja-Ling Wu. Scalable computation for spatially scalable video coding using NVIDIA CUDA and multi-core CPU. In *Proceedings of the 17th ACM international conference on Multimedia*, MM '09, pages 361–370, New York, NY, USA, 2009. ACM.

[205] P. Weaver L. Chan, J. W. Lee, A. Rothberg. H.264 in CUDA. Technical report.

[206] R. Rodriguez, J.L. Martinez, G. Fernandez-Escribano, J.M. Claver, and J.L. Sanchez. *Accelerating H.264 inter prediction in a GPU by using CUDA*. IEEE, January 2010.

[207] Erich Marth and Guillermo Marcus. Parallelization of the x264 encoder using OpenCL. In *ACM SIGGRAPH 2010 Posters*, SIGGRAPH '10, pages 72:1—-72:1, New York, NY, USA, 2010. ACM.

[208] Dishant Ailawadi, Milan Kumar Mohapatra, and Ankush Mittal. *Frame-based parallelization of MPEG-4 on compute unified device architecture (CUDA)*. IEEE, February 2010.

[209] Shang-Te Yang, Tsung-Kai Lin, and Shao-Yi Chien. Real-time Motion Estimation for 1080p videos on graphics processing units with shared memory optimization. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 297–302, 2009.

[210] Adobe Premiere Pro CS5.5. `http://www.adobe.com/products/premiere.html`.

[211] Elemental Technologies Inc. badaboom Media Converter. `http://www.badaboomit.com/`.

[212] CyberLink Corp. MediaEspresso 6.5. `http://www.cyberlink.com/products/mediaespresso/overview_en_US.html?&r=1`.

[213] Aleksander Gjermundsen. CPU and GPU Co-processing for Sound. (July), 2010.

[214] Xiaoliang Chen, Chengshi Zheng, Longhua Ma, Xiaobin Cheng, and Xiaodong Li. *Design and implementation of MPEG audio layer III decoder using graphics processing units*. IEEE, April 2010.

[215] FLACCL. `http://www.cuetools.net/wiki/FLACCL`.

[216] GStreamer: open source multimedia framework. `http://www.gstreamer.net`.

[217] Luis Reis. *A criação de redes de próxima geração usando produtos open source*. PhD thesis, 2008.

[218] Lian Wu and A.H. Aasgaard. *Migration of VOIP/SIP Enterprise Solutions towards IMS*. PhD thesis, Agder University College, 2006.

[219] Ram Kumar, Frank Reichert, Andreas Haber, Anders Aasgard, and Lian Wu. Migration of Enterprise VoIP/SIP Solutions towards IMS. *2007 3rd International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities*, pages 1–6, 2007.

[220] Fei Yao and Li Zhang. *OpenIMS and Interoperability with Asterisk / Sip Express VOIP Enterprise Solutions*. PhD thesis, Agder University College, 2007.

[221] The C10K Problem. `http://www.kegel.com/c10k.html`.

[222] Tornado Web Server. `http://www.tornadoweb.org/`.

[223] lighttpd. `http://www.lighttpd.net/`.

[224] nginx. `http://www.nginx.org`.

[225] Paul Tyma. Thousands of Threads and Blocking I/O. *SD West 2008*, 2008.

[226] Gu Liu, Hong An, Wenting Han, Guang Xu, Ping Yao, Mu Xu, Xiurui Hao, and Yaobin Wang. A Program Behavior Study of Block Cryptography Algorithms on GPGPU. *2009 Fourth International Conference on Frontier of Computer Science and Technology*, pages 33–39, 2009.

[227] Ricardo Portugal. transcodebench. http://github.com/rportugal/transcodebench.

[228] NVIDIA Corporation. OpenCL Best Practices Guide. 2011.

[229] Simple DirectMedia Layer. http://www.libsdl.org/.

[230] SDL mixer. http://www.libsdl.org/projects/SDL_mixer/.

[231] libsndfile. http://www.mega-nerd.com/libsndfile/.

[232] CppUnit. http://cppunit.sourceforge.net.

[233] SoX - Sound eXchange. http://sox.sourceforge.net/.